RT-JADE: Middleware com Suporte para Escalonamento de Agentes Móveis em Tempo-Real

Tatiana Pereira Filgueiras[1], Lau Cheuk Lung[1,2], Luciana de Oliveira Rech[2]

[1]Programa de Pós-graduação em Engenharia de Automação e Sistemas – PPGEAS
Universidade Federal de Santa Catarina (UFSC) – Florianópolis – Brasil

[2]Programa de Pós-graduação em Ciência da Computação – PPGCC
Universidade Federal de Santa Catarina (UFSC) – Florianópolis – Brasil
tati_tj@das.ufsc.br, {lau.lung, luciana.rech}@inf.ufsc.br

Resumo. *Para que um agente móvel com restrição temporal possa cumprir sua missão, é necessário que o mesmo cumpra 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 de tempo-real. Neste artigo adotamos um modelo de execução em que agentes móveis disputam um mesmo recurso em um mesmo host. Este artigo propõe o RT-JADE: uma extensão de middleware que possibilite agentes móveis concorrentes cumprirem suas missões, usando métodos de escalonamento em tempo-real sobre a plataforma JADE.*

Abstract. *For a mobile agent with a time restriction 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 paper, we adopt an execution model in which mobile agents compete for the same resource in the same host. The goal of this paper is to propose RT-JADE, a middleware extension that allows concurrent mobile agents to achieve their missions, using real-time scheduling methods.*

1. Introdução

Um Agente Móvel (AM) é um componente de software independente e autocontido, capaz de executar tarefas, e que não está limitado ao sistema onde iniciou a sua execução, ou seja, o agente 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. É um representante do conceito da mobilidade de código, a qual possui benefícios em sistemas distribuídos [1]: alto grau de flexibilidade, escalabilidade e customização; melhor utilização da infraestrutura de comunicação e a provisão de serviços de maneira autônoma sem a necessidade de conexão permanente. Por migrar para a localização do recurso desejado, um AM pode responder a estímulos rapidamente e pode continuar sua interação com o recurso se a conexão de rede cair temporariamente. Estas características fazem com que os AM 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.

Diversas pesquisas têm sido realizadas nos últimos anos no contexto de AM [2] [3] [4] [5]. Essas pesquisas têm evidenciado que o paradigma de agentes traz diversos benefícios ao projeto de sistemas distribuídos, pois encapsulam protocolos que facilitam a interoperabilidade entre plataformas, além de ter sua execução de forma assíncrona e autônoma. Em se tratando de aplicações com restrições temporais, os AM têm sido

adotados em diversas áreas tais como cuidados médicos[5], e-commerce [4], manufatura [6], além de ser uma alternativa para a construção de sistemas distribuídos [7].

Para facilitar o desenvolvimento de AM em sistemas distribuídos vários grupos de pesquisa desenvolveram plataformas de middleware [8] [9] específicas para esse fim. Os primeiros middlewares para AM foram desenvolvidos para redes LAN e WAN, tais como o *Grasshopper* [10], *Aglets* [11] e JADE [12]. E devido ao grande interesse em dispositivos móveis, foram propostos middlewares para redes de sensores sem fio [13] e dispositivos embarcados [8]. Estes últimos são projetados para dar suporte a mobilidade e comunicação de AM em redes sem-fio, considerando as limitações técnicas (bateria, processamento, memória, etc.) destes dispositivos.

Apesar da utilização de AM em sistemas de tempo-real já ser amplamente difundida e estudada, ainda não há pesquisas voltadas para o escalonamento de AM em aplicações de tempo-real, mais especificamente, o desenvolvimento de um middleware com suporte para escalonamento de AM. Um importante problema não tratado nas plataformas de middleware mencionadas anteriormente é o aspecto da concorrência dos recursos de um *host* remoto. Quando vários agentes visitam esse *host* simultaneamente, para utilizar um determinado recurso, não há mecanismos de escalonamento tempo-real para o uso deste recurso por parte dos agentes visitantes. Ou seja, o que normalmente se faz é o atendimento destes agentes segundo a ordem de chegada. Em aplicações tempo-real, AM devem executar suas tarefas de acordo com suas restrições temporais (*deadline*s) e, baseado nisso, é necessário atribuir prioridades no escalonamento para a utilização dos recursos, deste modo, permitindo aos agentes mais prioritários utilizar os recursos mais rapidamente.

A busca por soluções para desenvolver um sistema de escalonamento de AM é de extrema importância para aplicações que necessitam cumprir restrições temporais, pois tais agentes são normalmente assíncronos em relação à chegada e saída dos *host*s devido a 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 [12]. Um exemplo é o contexto de uma oficina de produção, onde todos os recursos de manufatura de peças são frequentemente representados como AM, enquanto outros recursos de produção, como máquinas, instrumentos e ferramentas de corte são representados como agentes estáticos [6]. 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 pesquisa, propomos a arquitetura RT-JADE, que tem como objetivo unir a mobilidade de código com a questão temporal, tratando a questão de escalonamento de AM com restrições temporais, visando a questão de concorrência entre estes. Este artigo apresenta uma proposta de extensão da plataforma para AM JADE para dar suporte a escalonamento de AM. As extensões de middleware propostas comportam diferentes tipos de políticas de escalonamento, como: FIFO (*First In First Out*), LIFO (*Last In First Out*), EDF (*Earliest Deadline First*), EDF preemptivo, Prioridade, Prioridade preemptivo, *Deadline Monotonic*, *Deadline Monotonic* preemptivo, SJF (*Shortest Job First)* e SRTF (*Shortest Remaining Job First)*, oferecendo uma ampla gama de políticas de escalonamento para diferentes necessidades de aplicações tempo-real usando AM. Para 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.

2. Trabalhos Relacionados

No campo da computação distribuída inteligente em redes de computadores dinâmicas, uma questão que atrai interesse é o uso de AM e Sistemas MultiAgentes (SMA). Em [4] foi proposto um algoritmo utilizando agentes centralizados para auxiliarem dois níveis distintos de AM: *Broker Level Mobile Agent* (BMA) e *Supplier Level Mobile Agent*. O objetivo da proposta é a redução do tempo de completude da missão por parte dos AM 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, o clone encaminha o resultado ao BMA, que por sua vez, elege a melhor dentre as respostas recebidas e a encaminha ao agente centralizador. O clone proprietário da melhor resposta conclui a compra do produto, enquanto os demais se autodestroem. Ao concluir a compra, o clone encaminha o histórico de operações ao BMA que o reencaminha ao agente centralizador.

A utilização do middleware *Agilla* para facilitar a comunicação inter-agentes em redes de sensores foi proposta por [13] com o objetivo de facilitar a comunicação entre AM por utilizar um espaço de tuplas. Em [5] foi proposto um middleware tempo-real utilizando um protocolo de mensagens móveis confiável através de uma rede wireless para transmissão de dados. A ideia principal é que um profissional de saúde possa enviar informações de determinado paciente através de mensagens (encaminhadas por um AM) por dispositivos móveis, como PDA, para o hospital para que o paciente possa ser atendido da melhor forma possível.

Em [14] é proposto um módulo que permite ao JADE fornecer tolerância a faltas e segurança em sistemas multiagente. O autor cita alguns recursos, como detecção de ação maliciosa no *host* remoto e 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. O projeto MRSCC [2] visa produzir uma arquitetura tempo-real integrada ao JADE para possibilitar a criação de produtos com AM para membros da cadeia de abastecimento. Cada vez que um usuário loga-se, um agente dedicado é criado para atender seu pedido. Ele migra à procura de produtos e do melhor preço, e tem a capacidade de efetivar a compra. Porém, os autores não apresentaram de forma clara a utilização de um mecanismo de tratamento para requisições concorrentes em tempo-real.

O presente artigo é o primeiro na literatura, do nosso conhecimento, a abordar o tratamento de concorrência quanto à utilização de um recurso de uso exclusivo em determinado *host*. Seguindo a visão de [14], neste artigo é proposta uma melhoria à plataforma JADE que permita o tratamento de requisições de AM concorrentes tempo-real por meio de um middleware. Para simulação deste middleware, montou-se uma arquitetura baseada no modelo de [4] com utilização de um *Broker* e agentes estáticos em cada nodo para atender às requisições dos AM e gerenciar os recursos disponíveis no nodo. Outro objetivo do presente artigo é a atribuição dinâmica de prioridades para os AM através de algoritmos de escalonamento.

3. JADE e Tempo-Real

JADE (*Java Agent DEvelopment Framework*) [12] é um *open-source* estruturado para tornar o desenvolvimento mais rápido de aplicações multi-agente, em conformidade

com as especificações FIPA (*Fundation for Intelligent Physical Agents*). O JADE pode ser considerado um middleware que implementa um framework de desenvolvimento e uma plataforma de agentes, acompanhando um conjunto de bibliotecas, para desenvolvimento de agentes na linguagem Java. A arquitetura da plataforma JADE é baseada na coexistência de várias máquinas virtuais Java (JVM) podendo ser distribuída por diversas máquinas independentes, com sistemas operacionais distintos. Cada plataforma JADE possui um ou mais containers, porém, todas as plataformas possuem o *main-container,* que fornece o AMS (*Agent Management System*) responsável pelo endereçamento da plataforma, entrega e envio de mensagens, criação/destruição e recebimento de agentes; o DF (*Directory Facilitator*) que oferece o serviço de páginas amarelas na plataforma e o registro RMI, para registrar e recuperar referências a objetos (agentes) através de seus nomes. A comunicação entre as plataformas dá-se através de invocação remota de métodos, ou RMI da linguagem Java (Figura 1).

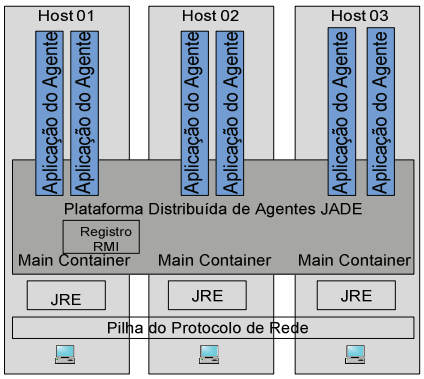


Figura 1. Arquitetura Distribuída de uma Plataforma de Agentes JADE

No JADE, um agente é autônomo, um processo independente que possui identidade e requer comunicação com outros agentes, seja ela por colaboração ou por competição, para completar seus objetivos [12]. Cada agente JADE é tratado como uma *thread* independente de execução que emprega múltiplas tarefas ou comportamentos e conversações simultâneas, e possui uma fila privativa de tamanho finito, criadas e armazenadas pelo subsistema de comunicação do JADE, projetado para conseguir o menor custo na troca de mensagens.

JADE também suporta mobilidade de agentes em uma plataforma que pode ser distribuída, possuindo distintos Sistemas Operacionais e as configurações podendo ser controladas via uma *Graphical User Interface* (GUI) remota. Um AM é transportado por meio de um *Java Archive* (JAR), que contém o estado serializado do agente, entre outros dados. Sua configuração pode ser alterada em tempo de execução, movendo agentes de uma máquina a outra, quando necessário. Um agente deve ser capaz de realizar várias tarefas simultâneas em resposta a diferentes eventos externos. O JADE dá suporte a paralelismo, máquina de estados finitos, comportamento atômico, sequência e concorrência apenas entre os comportamentos (ou *behaviours)* do próprio agente, porém, o tratamento de mensagens trocadas entre agentes, bem como o de chegada de novos AM a um determinado *host* é feito pela política de escalonamento FIFO.

3.1 Conceitos de Escalonamento Tempo-Real e Concorrência

Define-se como concorrência quando dois ou mais processos solicitam a utilização de um mesmo recurso simultaneamente [15], sendo um elemento muito importante da programação distribuída. Há dois tipos de interações entre processos concorrentes [16],

sendo elas: competição (quando processos são independentes, mas têm de partilhar os recursos comuns de um computador: tempo de CPU, espaço de memória, acesso aos periféricos); e cooperação (quando processos são interdependentes, fazendo parte de uma mesma aplicação, ou interagindo explicitamente uns com os outros).

Para resolver questões de concorrência entre processos, faz-se uso de técnicas de escalonamento. O escalonador é uma rotina que gerencia o compartilhamento de processadores ou sistemas distribuídos entre tarefas, dando a cada tarefa em execução uma fatia de tempo do processador, priorizando determinados tipos de processos [15]. Basicamente, o escalonador é quem define a ordem de execução entre os processos candidatos. Os principais objetivos de um algoritmo de escalonamento são: Justiça, *Policy Enforcement* e Equilíbrio [17]. O escalonamento pode ser preemptivo e não preemptivo, podendo adotar um dos seguintes critérios [18]: orientados a desempenho (*turnaround time*, tempo de resposta, *deadline* ou predictabilidade); e orientados ao sistema (utilização da CPU, *throughput, fairness*, prioridades, balanceamento de recursos ou tempo de espera).

Na política de *best-effort*, a estratégia de alocação dinâmica de recursos baseia-se em estudos probabilistas (ou históricos anteriores) sobre a carga esperada e os cenários de falhas aceitáveis. Portanto, os tempos de chegada das tarefas não são conhecidos previamente (carga dinâmica). Essa abordagem deve lidar com situações, em tempo de execução, onde recursos computacionais são insuficientes para os cenários de tarefas que se apresentam (situações de sobrecarga). No escalonamento *best effort* não há garantias de que todos cumpram seus *deadline*s em tempo de projeto, porém, o sistema fará o possível para tal. A possibilidade de sobrecarga no sistema é uma das consequências da abordagem. Para tratar esta sobrecarga, abordagens como descarte de tarefas ou execução de todas, mas com sacrifício do *deadline* de algumas, são empregadas, porém, é necessária a definição prévia sobre quais tarefas podem ser sacrificadas.

4. RT-JADE : Middleware para Agentes Móveis Tempo-Real

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

4.1 Modelo de Execução de Agentes Móveis Tempo-Real

Neste trabalho, consideramos um conjunto de computadores conectado em uma rede. Cada *host* dessa rede possui o middleware RT-JADE onde os AM podem migrar, executar sua missão e, por fim, partir para outro *host* (Figura 2). Uma missão é composta por um conjunto de recursos que devem ser executados. Um itinerário é definido como uma sequência de *host*s que um agente deve visitar para consumir estes recursos e, assim, cumprir a missão de uma aplicação. Cada *host* é capaz de receber a quantidade de AM que puder, limitada apenas pela sua capacidade de memória e processamento. Portanto, para aplicações 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 *deadline*s mais próximos, mesmos chegando mais tarde, podem pular para o início da fila.

A interação entre cliente e servidor é efetuada através da utilização de AM, porém, de forma transparente ao usuário. Assume-se, por simplicidade, que cada *host* possui um

único recurso, e que tal recurso seja de uso exclusivo ao AM 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, adotamos o uso de "*visões"* (*view*) que indica o conjunto de AM em um *host* em um determinado instante. 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 destes agentes. Durante a $visão_i$ ($view_i$), agentes escalonados podem desistir de esperar pelo recurso e, como consequência, desistir da fila deixando o *host* (migrar para outro *host* menos "sobrecarregado" ou voltar para o *host* origem), a desistência de um *host* ocorre quando o agente verifica a impossibilidade de concluir sua missão dentro do seu *deadline*. Durante a $visão_i$, pode ocorrer também a entrada de novos AM no *host* (agente em espera no *host* 3 da Figura 2) – esses agentes ficam fora da $visão_i$ e, portanto, não são escalonados para usar o recurso no momento.

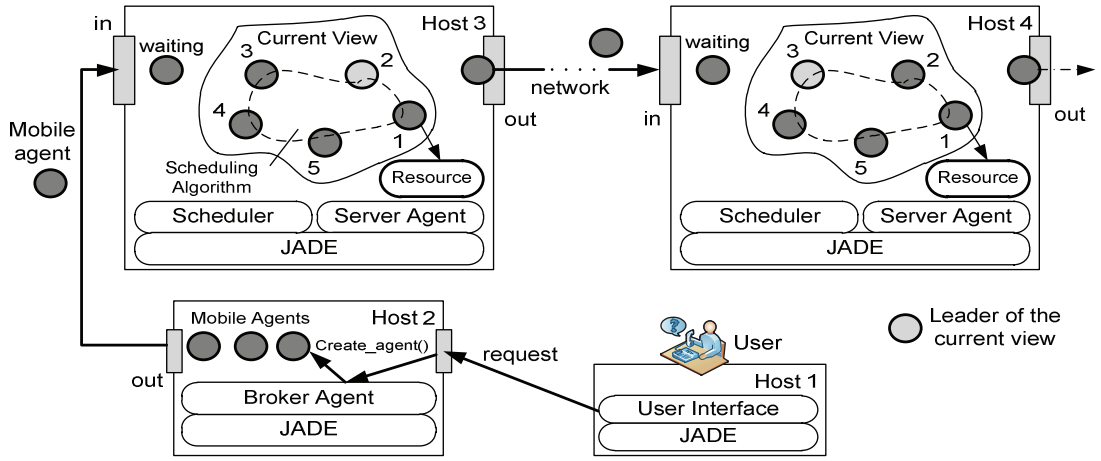


Figura 2. Modelo de execução da Arquitetura RT-JADE

O algoritmo de escalonamento (*host* 3 da Figura 2) é executado uma única vez para uma determinada *visão*. Uma mudança de visão (para $visão_{i+1}$) pode ocorrer quando houver a entrada de novos agentes no *host*. A verificação se há novos agentes em espera para gerar a nova visão (a $visão_{i+1}$) é feita a cada vez que um agente na $visão_i$ terminar de utilizar o recurso e deixar o *host*. Neste ponto, se houver pelo menos um agente em espera, uma nova visão ($visão_{i+1}$) é estabelecida e o algoritmo de escalonamento é executado para definir a nova ordem nesta 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 desta fila.

4.2 Arquitetura RT-JADE

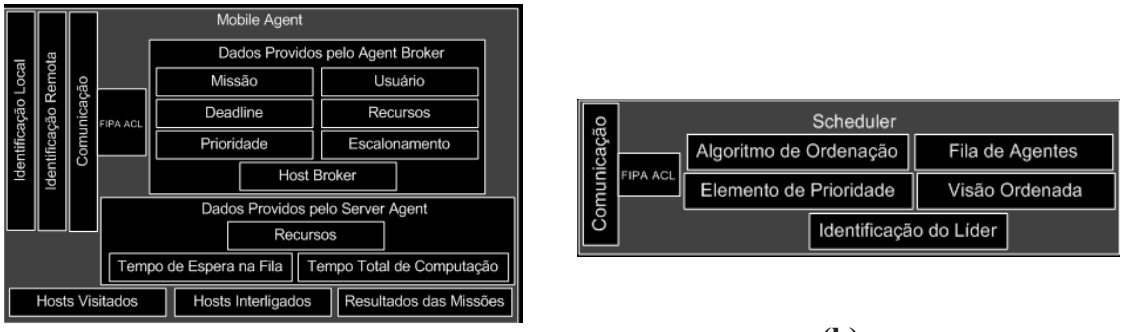
A arquitetura consiste basicamente de três agentes estacionários (*User Interface*, *Broker Agent* e *Server Agent*) e dos AM visitantes (Figura 2). O sistema como um todo possui apenas um *Broker* e até ***n*** *Server Agents*. Os agentes, bem como o *Scheduler* rodam sobre a plataforma JADE. A função da *User Interface* (lado cliente) é enviar e receber respostas *ACLMessage*, no padrão FIPA ACL, do agente estacionário *Broker*.

A função do *Broker Agent* é receber requisições dos usuários (*host* 1 e *host* 2 - Figura 2) e criar um AM para cada tupla de requisições recebida. Note que um usuário pode ter mais de uma requisição dentro da tupla de requisições, porém, nunca mais de um AM (salvo casos em que o usuário faz outra requisição após a primeira enviada ao *Broker*). O *Broker* também define os elementos de prioridade (*deadline* e credencial) dos AM

conforme o tipo de escalonamento utilizado. O *Agente Broker* tem também por finalidade aguardar o retorno dos AM e informar os resultados da missão ao usuário da aplicação através da *User Interface*.

O *Server Agent* é uma interface para administração dos recursos oferecidos pelo *host*. Tem como função o recebimento de AM, a comunicação com estes (para fornecimento de tempo estimado de espera para utilizar o recurso), a escolha de um líder para a *visão* dentre os AM contidos na mesma e provê o acesso ao recurso exclusivo. Sua estrutura interna (Figura 3) é composta de uma Identificação Remota (provida pelo JADE, que permite agentes remotos se comunicarem), uma Identificação Local (provida pelo JADE, que permite agentes locais se comunicarem), uma interface de Comunicação com especificações da FIPA (provida pelo JADE, que permite que agentes que sigam as especificações FIPA se comuniquem, independente de que plataforma ou linguagem de programação sejam implementados), a lista de Recursos disponíveis no *host*, uma Fila de Agentes para comportar os AM no *host*, um *Membership Service* para fins de tratamento de visões e a lista de *Hosts* Interligados ao *host* corrente. Além destes, o *Server Agent* possui um Servidor de Cálculos instanciado, que contém a Identificação do Banco de Dados para fins de acesso (leitura, escrita, atualização, etc.) ao histórico e outros tipos de dados.

Figura 3. Estrutura do Server Agent



(a) **(b)**

Figura 4. (a) Estrutura do Agente Móvel e (b) Estrutura do *Scheduler*

Além desses agentes, o sistema ainda dispõe dos AM criados pelo *Broker* e do *Scheduler*. Um AM tem como função migrar para os *host*s a fim de completar uma missão. Uma missão só é dada como completa se for cumprida sem ultrapassar o seu *deadline*. O agente tem também como função retornar ao *Broker* e informá-lo do resultado da missão. Sua estrutura interna (Figura 4(a)) consiste de uma Identificação Remota, uma Identificação Local, uma Interface de Comunicação com especificações da FIPA, uma lista de *Host*s Visitados, uma lista de *Host*s Interligados ao *host* e o Resultado da Missão. Além desses, o AM possui dados fornecidos pelo *Broker* como: uma missão a ser cumprida, a identificação do usuário que requisitou tal missão, um *deadline* para completude da missão, uma lista de recursos necessários para concluir a

missão, uma credencial de prioridade, a identificação da política de escalonamento a ser empregada e o endereço do *Host Broker*. Possui também alguns dados providos pelo *Server Agent*, como a lista de recursos disponíveis no *host*, o tempo de espera na fila (estimado) para poder utilizar o recurso e o tempo total (estimado) de computação a ser gasto pelo AM para utilizar todos os recursos necessários pelo mesmo no *host*.

Por fim, o *Scheduler* (*host* 3-Figura 2) ordena os AM da visão, de acordo com a política de escalonamento empregada, e fornece ao *Server Agent* a identificação do AM mais prioritário para utilizar o recurso. O *Scheduler* utiliza a comunicação FIPA ACL do JADE, e possui como identificação remota e local a mesma do AM que o instanciou (Figura 4(b)). O escalonador recebe do AM a visão corrente e insere em sua Fila de Agentes pessoal. Cada escalonador possui um elemento de prioridade distinto (ex.: DM= *deadline*; PRIO=credencial) ao qual efetuará a ordenação com base neste e guardará a Visão Ordenada para informá-la ao servidor posteriormente. Além disto, o *Scheduler* contém a identificação do AM Líder, ou seja, o AM que o instanciou; e o algoritmo de ordenação (*FAst Quick Sort*) necessário para ordenar a visão.

4.3. Algoritmo de Escalonamento Tempo-Real dos Agentes Móveis

Conforme explicado na seção 4.1, o escalonamento é feito apenas para os agentes pertencentes a visão atual ($visão_i$). Na visão atual, o *Server Agent* elege aleatoriamente um dos AM contidos na visão para ser o líder da mesma. O AM eleito passa a ter a responsabilidade sobre o escalonamento de si e dos demais AM 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(a), o agente líder (*Leader*) requisita o elemento de prioridade a **todos** os AM (A1, A2, ..., An) contidos na visão atual. Após receber as respostas, seleciona o AM com maior prioridade, baseado na política de escalonamento empregada. Então, o agente mais prioritário recebe autorização para utilizar o recurso exclusivo (neste exemplo, o AM A2 é o mais prioritário) e o *Leader* informa ao *Server Agent* a identidade do mesmo. Após o AM terminar sua missão no *host* corrente, um novo *Leader* é escolhido para a próxima visão.

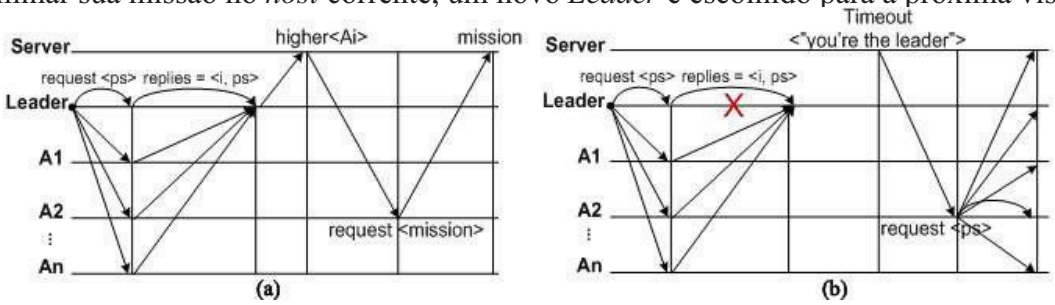


Figura 5. Escalonamento sem falha (a) e com falha (b) do *Leader*

Caso ocorra de um *Leader* sofrer um *crash*, outra visão é iniciada e um *Leader* distinto é escolhido para esta nova visão, evitando assim, *crash* em todo o sistema. A Figura 5(b) ilustra o comportamento do *Server*, onde o *Leader* sofreu um *crash*. Se até o timeout o *Server* não receber a resposta do *Leader* atual, o mesmo elege aleatoriamente um novo *Leader* para iniciar uma nova visão. Caso a visão não mude, ou seja, não houve a entrada/preempção de AM no *host*, o escalonamento não será necessário. Então, o *Leader* da visão apenas comunica ao *Server* o próximo AM a ter permissão de utilizar o recurso.

Os algoritmos de escalonamento não preemptivo e preemptivo são apresentados na Figura 6. As caixas de texto da figura são adições no algoritmo não preemptivo para

torná-lo um algoritmo preemptivo. As notações utilizadas nestes algoritmos são apresentadas na Tabela 1. O funcionamento do algoritmo executado pelo *Server Agent* é descrito nas linhas 5-22. Quando o *host* recebe um AM, o *Server Agent* o enfileira e define como verdadeira a variável *newView* (linhas 6 a 8). Se o AM deixar o *host* antes de utilizar o recurso, o mesmo é retirado da fila e *newView* passa a ser verdadeiro (linhas 12 a 14). Caso um recurso já esteja sendo utilizado, quando o mesmo é liberado, o *Server Agent* guarda na variável *preview_view* a visão anterior, e atualiza a variável *current_view* com a visão atual, ou seja, a fila atual de AM (linhas 15 a 17). Após isso, é realizada uma nova escolha de Líder para a próxima visão (linha 18). Como o escalonamento será executado, a variável *newView* é definida como falsa, até que um próximo AM chegue ao *host* (linha19). Com isso, sempre que um AM liberar o recurso e nenhum outro agente tiver migrado ao *host*, a variável será falsa, garantindo assim, no código do *Mobile Agent*, que a função *newView( )* retorne falso. Mas, caso a visão seja a primeira, ou seja, o algoritmo esteja sendo executado pela primeira vez e ainda não haja Líder, ele será eleito (linhas 20 a 22).

O algoritmo do *Mobile Agent* (linhas 23-31) consiste em verificar se o mesmo é o líder através do recebimento da mensagem "*you're the leader*" (linha 26) enviada pelo *Server Agent* através da função *chooseLeader*. Caso seja o Líder, verifica 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 27). Caso uma das duas situações anteriores seja verdadeira, o *Mobile Agent* cria uma instância da classe *Scheduler* (linha 28). Caso ambas sejam falsas, o escalonamento não será necessário por não ter havido alterações no escalonamento anterior, então, se o recurso estiver livre, o AM com maior prioridade é enviado à sessão crítica (linhas 29 a 31).

O *Scheduler* (linhas 32-42) por sua vez, requisita de cada AM da fila (inclusive o que o instanciou) seus respectivos objetos de prioridade p_s (linhas 33-34), que podem ter somente o *deadline; deadline* e credencial; ou somente credencial para o caso do escalonamento de prioridades onde o *deadline* só é utilizado para fins de avaliação de desempenho. Após isto, o *Scheduler* aguarda as respostas dos agentes por um determinado período t_1, e tais respostas são inseridas em uma fila (linhas 35-38). Caso haja timeout ($t_1 > t_0$), a variável *replies* terá valor nulo, e este valor estará enfileirado na posição correspondente ao AM que deixou de responder dentro do tempo previsto. O escalonamento é então executado ignorando os valores nulos (se houverem) e o AM com maior prioridade é enviado ao recurso exclusivo (linhas 40-42). A ordenação dos valores recebidos, feita pela função *schedule()*(linha 40), é feita utilizando o algoritmo *Fast Quick Sort* [19]. Para o escalonamento FIFO, o Líder requisita apenas ao Agente Servidor que envie o próximo AM da fila ao recurso exclusivo.

O funcionamento do algoritmo de escalonamento preemptivo é semelhante ao não preemptivo, com exceção das linhas:

- 9 a 11 do *Server Agent*, onde o mesmo verifica no momento da chegada de um AM se há algum outro utilizando o recurso. Em caso afirmativo, envia o endereço do agente que utiliza o recurso (que chamaremos de A_j) para o agente que chegou (que chamaremos de A_i), para que o mesmo tente uma possível preempção;
- 24 a 25 do *Mobile Agent,* onde o mesmo recebe o endereço de A_j no momento e tenta a preempção com tal;
- 41 do *Scheduler*, onde é realizada a mesma verificação das linhas 29-31 do *Mobile Agent*;

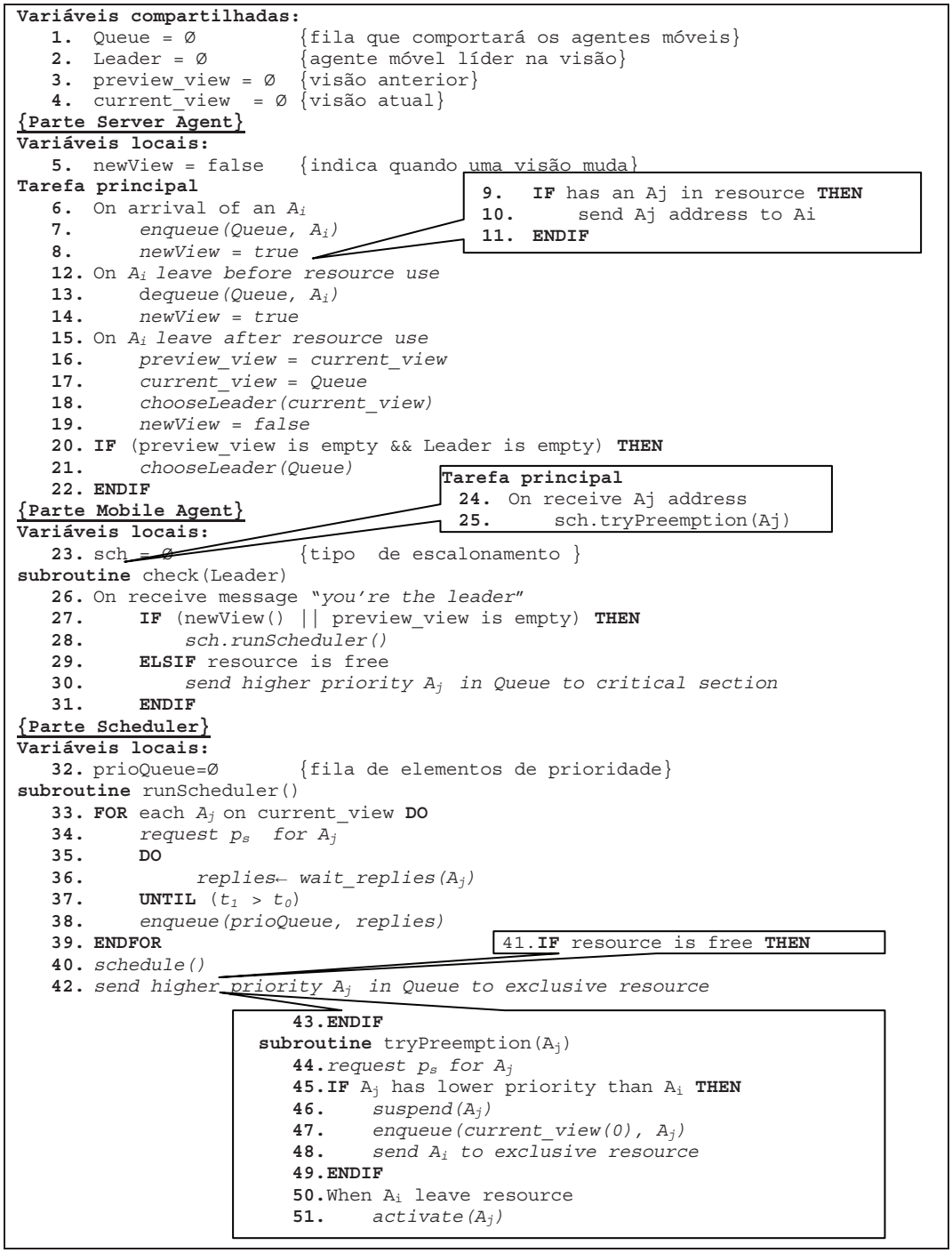


Figura 6. Algoritmo de Escalonamento Não Preemptivo e Preemptivo

Tabela 1. Notações

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

Nas linhas 44 a 51 é onde o algoritmo de preempção é em si descrito. A_i se comunica com A_j solicitando seu objeto de prioridade p_s (linha 44). Caso A_j seja menos prioritário que A_i, o mesmo é suspenso e reinserido no início da visão, ou seja, da fila *current_view* (linhas 45-47). Após isso, A_i passa a utilizar o recurso (linha 48). Ao terminar, o agente suspenso A_j é reativado, voltando assim a utilizar o recurso do ponto de onde parou (linhas 50-51). As funções de suspensão e reativação de um AM são fornecidas pelo JADE, através dos métodos *suspend()* e *activate()*.

5. Implementação e Testes

Neste trabalho, foram analisados dez algoritmos de escalonamento, destes, nove implementados e um (FIFO) provido pelo JADE: FIFO (*First In, First Out*); EDF (*Earliest Deadline First*) não preemptivo; EDF preemptivo (PEDF); LIFO (*Last in, First Out*); Escalonamento baseado em prioridade não-preemptivo (PRIO); Escalonamento baseado em prioridade preemptivo (PPRIO); *Deadline Monotonic* não preemptivo (DM); *Deadline Monotonic* preemptivo (PDM); SJF (*Shortest Job First*); SRTF (*Shortest Remaining Time First*) ou SJF preemptivo. A descrição das políticas de escalonamento é descrita em [20] e [21].

Toda a arquitetura (AM e Estacionários, *Broker* e Algoritmos de Escalonamento) foi implementada na linguagem JAVA (JDK 1.6.0_19), utilizando o framework JADE (versão 4.0.1). 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; (*iii*) *Intel Core Quad* 3.0GHZ, 4GB RAM, Windows 7 64 bits.

Para este artigo, foi simulado dois *Server Agent*, cada um contendo apenas um único recurso, conforme mencionado anteriormente na seção 4.1. O *Server Agent* do *host* 1 possui o recurso de busca em base de dados, enquanto o *Server Agent* do *host* 2 possui o recurso de cálculos gerais, como preço de peças, dimensões, orçamento, etc. A configuração dos nodos é por ordem de precedência, ou seja, o AM deve passar primeiro pelo *host* 1 para então chegar ao *host*2, assim, não sendo possível utilizar o recurso R2 sem antes passar por R1 (Figura 7). As faixas de *deadline* foram escolhidas baseadas em históricos de simulações anteriores e carga computacional, o que justifica a escolha distinta de *deadline*s por quantidade de AM concorrentes. Cada AM possui de 1 a 3 missões distintas entre si para serem utilizadas por ambos os recursos. Tais missões são escolhidas aleatoriamente, portanto, há a possibilidade de mais de um AM possuir uma mesma missão.

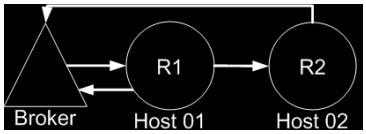


Figura 7. Configuração dos nodos para simulação da arquitetura

Para efeitos de teste, foram executadas 100 iterações para cada escalonamento em cada faixa de *deadline*, totalizando assim 500 AMs para cada escalonamento por faixa de *deadline* em IT500 e 2000 AMs em IT2k. Em IT500, 5 AMs concorrentes são lançados por iteração, enquanto que em IT2k, 20 AMs concorrentes são lançados. O intervalo de cada iteração não ultrapassa 1 segundo após o Broker ter recebido todos os AMs da iteração anterior. Para efeitos de análise, foram medidas as quantidades de AMs de cada escalonamento que cumpriram totalmente (consome R1 e R2) ou parcialmente (consome apenas R1) suas missões bem como o número dos que não cumpriram. Um

consumo parcial é quando um AM, que terminou de consumir o recurso R1, percebe que seu deadline está muito próximo e, portanto, desiste de R2, retornando ao *Broker* e assim, completando parcialmente sua missão.

A medição do tempo de viagem do AM começa "a correr" a partir do momento em que o AM sai *Broker* (Figura 2 e 7). O AM deve sair do *Broker* e 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, migrar de volta ao *Broker* antes do deadline.

Com as medidas, foram comparados os algoritmos 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.

5.1 Resultados Obtidos

O código de um agente móvel é compactado em um arquivo JAR (*Java Archive*) com 13 Kbytes de tamanho. A migração de um AM de um host para outro é realizada através da transmissão deste arquivo. No JADE, o tempo de transmissão deste arquivo ficou em torno de 83,5*ms*. Em relação ao consumo dos recursos R1 e R2, os agentes levaram, em média, 6,2*ms* para consumir o recurso R1 e 9,2ms para consumir o recurso R2.

Para as simulações IT500 os resultados são demonstrados na Figura 8. Analisando-os, é possível concluir que para *deadline*s mais apertados (300-500ms) o algoritmo de escalonamento LIFO apresentou melhor desempenho em ambos os casos, comparando completude total da missão, sendo que na faixa de 300ms, o escalonamento menos indicado foi o SJF que ocasionou na perda total da missão para todos os AM. Para a faixa de 700ms os algoritmos PEDF, PRIO e PPRIO apresentaram os melhores resultados, sendo o SJF e seu preemptivo SRTF os piores, enquanto que com *deadline*s mais folgados (1100ms) todos os algoritmos apresentaram resultados satisfatórios, sendo os piores DM e FIFO. É importante salientar que com base na população de AM que não cumpriram sua missão, quer total quer parcialmente, o algoritmo FIFO ficou entre os piores resultados na maioria das faixas de *deadline* escolhidas.

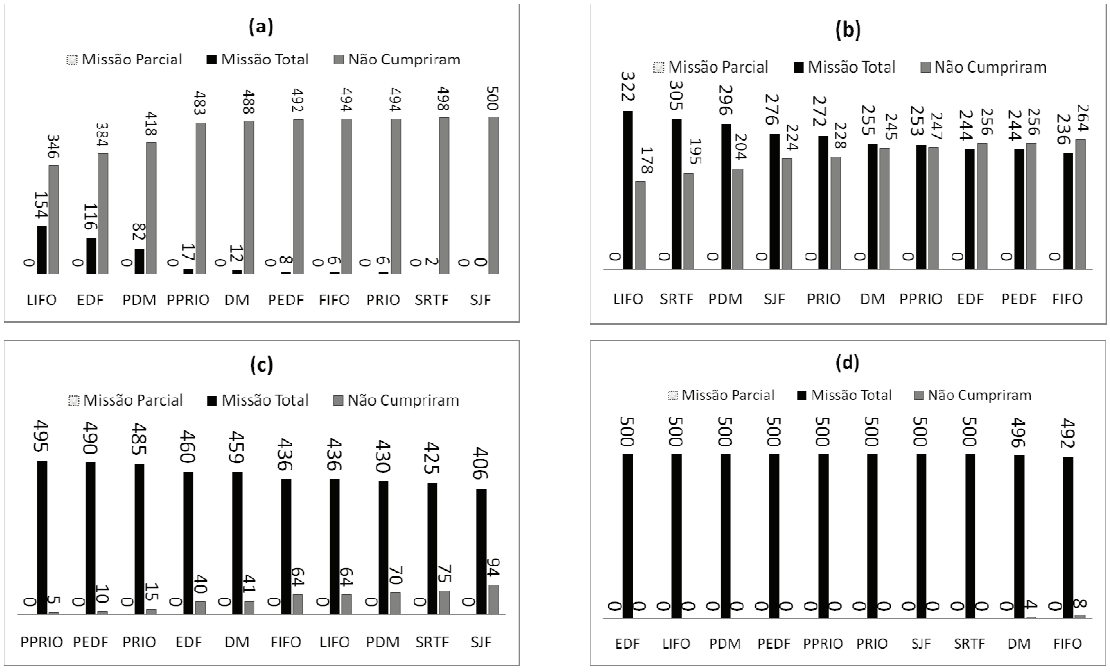


Figura 8. Cinco AM concorrentes com faixa de *deadline* de 300 (a), 500 (b), 700 (c) e 1100ms (e)

Para as simulações IT2k, os resultados obtidos são demonstrados na Figura 9. Para a faixa de *deadline* de 300ms todos os agentes não conseguiram cumprir sua missão, portanto, os resultados foram desprezados e foi escolhida uma nova faixa. Para a faixa de *deadline* de 700ms todos os algoritmos apresentaram baixo desempenho, enquanto que para as faixas de 1500ms a 2000ms os algoritmos apresentaram uma grande melhora, sendo os mais indicados para 1500ms PRIO, EDF e PPRIO e para 2000ms PEDF, PDM e EDF, podendo assim observar que o algoritmo EDF está entre os mais indicados para ambos os casos. Na faixa mais folgada (2500ms) os algoritmos que permitiram que todos os AM cumprissem suas missões foram o DM e o PEDF. Para esta simulação, a política FIFO foi a de pior resultado em três das quatro faixas escolhidas.

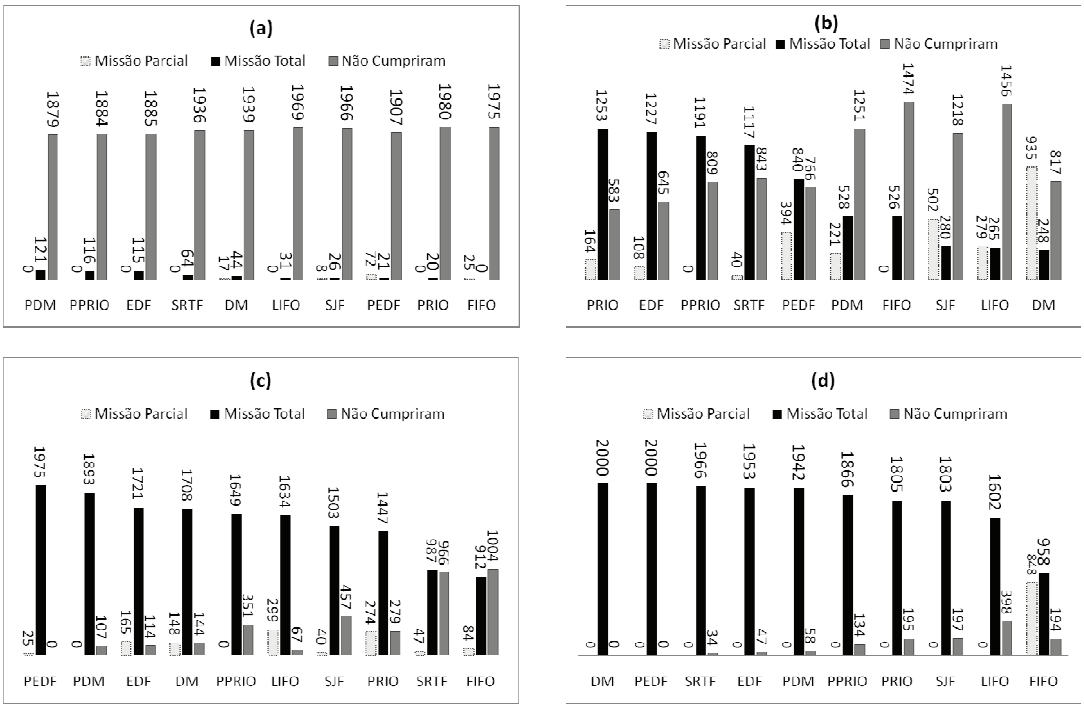


Figura 9. Vinte AM concorrentes com faixa de *deadline* de 700 (a), 1500 (b), 2000(c) e 2500ms (d)

6. Conclusões

Este artigo apresentou uma extensão de middleware que fornece ao JADE suporte a tempo-real, baseado no conceito de visões. A proposta foi testada no próprio framework JADE e medidas de desempenho mostraram que, a quantidade de AM com restrições temporais que cumpriram sua missão dentro do *deadline* estipulado utilizando o RT-JADE foi de 73,3 a 100% maior para cinco AM concorrentes e de 41,98 a 100% maior para vinte AM concorrentes, em relação à atual política de escalonamento (FIFO) fornecida pelo JADE.

Com a realização deste trabalho, algumas perspectivas para melhorias poderão vir a ser desenvolvidas, como: cálculo de tempo de espera para AM utilizando média ponderada, dada por cálculos baseados em Teoria das Filas; otimização dos algoritmos

de escalonamento preemptivos e a adição de novas políticas de escalonamento, como *Round Robin*. Esses resultados serão reportados em trabalhos futuros.

7. Referências

[1] Fou, J. (2010), "Web Services and Mobile Intelligent Agents - Combining Intelligence with Mobility," Disponível em: http://www.webservicesarchitect.com/content/articles/fou02.asp.

[2] Shemshadi, A.; Soroor, J. and Tarokh, M. J. (2008) "Implementing a Multi-Agent System for the Real-time Coordination of a Typical Supply Chain Based on the JADE Technology," IEEE International Conference on System of Systems Engineering (SoSE '08), pp. 1–6.

[3] Baek, J., Kim G. and Yeom, H. (2002) "Cost-Effective Planning of Timed Mobile Agents", International Conference on Information Technology: Coding and Computing (ITCC'02).

[4] Sahingoz, O. K. and Erdogan, N. (2004), "A Two-Leveled Mobile Agent System for E-commerce with Constraint-Based Filtering," LNCS, Springer-Verlag, vol. 3036 (ICCS 2004), pp. 437–440.

[5] Arunachalan, B. and Light, J. (2008) "Agent-based Mobile Middleware Architecture (AMMA) for Patient-Care Clinical Data Messaging Using Wireless Networks". Proceedings of 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications.

[6] Wang, S.; He, D. and Goh, M. W. T. (2006) "An Intelligent Manufacturing System: Agent Lives in Adhesive Slice," International Journal of Computer Science and Network Security, vol.6 , no.5A.

[7] Ca, J.; Wang, X.; Lo, S. and Das, K. (2002) "A Consensus Algorithm for Synchronous Distributed Systems using Mobile Agent", Proceedings of the PRDC.

[8] Chen, B.; Cheng, H. and Palen, J. (2006), "Mobile-C: A Mobile Agent Platform for Mobile C/C++ Code," Software - Practice & Experience, Vol. 36, Issue 15, pp. 1711-1733.

[9] Wierlemann, T.; Kassing, T. and Harmer, J. (1997). "The OnTheMove Project: Description of Mobile Middleware and Experimental Results", Springer Book Series, Vol 435, pp.21-35.

[10] Bäumer, C.; Magedanz, T. (1999), "Grasshopper - A Mobile Agent Platform for Active Telecommunication," IATA '99 Proceedings of the 3th Int. Workshop on IATA, pp. 19-32.

[11] Lange,D. B.; Oshima, M.; Karjoth, G. and Kosaka, K. (1997), "Aglets: Programming mobile agents in Java, " In WWCA, Vol. 1274, pp. 253-266.

[12] Caire, G.; Bellifemine, F.; Greenwood, D. (2007), "Developing Multi-Agent Systems with JADE", (Wiley Series in Agent Technology). vol. 1 , pp. 10-113. ISBN: 978-0-470-05747-6.

[13] Fok, C.; Roman, G. and Lu,C. (2006), "Mobile Agent Middleware for Sensor Networks: An Application Case Study", Information Processing in Sensor Networks, pp. 382 – 387.

[14] Leung,K.K, (2010). "FTS Framework for JADE", em: http://www.cse.cuhk.edu.hk/~kwng/FTS.html.

[15] Barland, I.; Greiner, J. and Vardi, M. (2005), "Concurrent Processes: Basic Issues", [Connexions Web site]. October 6, 2005. Disponível em: http://cnx.org/content/m12312/1.16/.

[16] Shrivastava, S. K. and Banatre, J. P. (1978), "Reliable Resource Allocation Between Unreliable Processes," IEEE Transactions on Software Engineering, vol. 4, no. 3, pp. 230–241.

[17] Chang, J.; Zhou,W.; Song, J. and Lin, Z. (2010), "Scheduling Algorithm of Load Balancing Based on Dynamic Policies", ICNS'10 Sixth International Conference on Networking and Services, pp. 363-367.

[18] Choi, S.; Kohout, N.; Pamnani, S.; Kim, D. and Yeung, D. (2004), "A general framework for prefetch scheduling in linked data structures and its application to multi-chain prefetching," ACM Transactions on Computer Systems, 22(2):214–280.

[19] Ahrens,D.(2005),"Fast Quick Sort for JAVA," Disponível em: http://people.cs.ubc.ca/~harrison/Java.

[20] Farines, J.M; Fraga, J. S.; Oliveira, R. S. (2000), "Sistemas de Tempo Real," Escola de Computação 2000. Departamento de Ciência da Computação USP.

[21] K. Ramamrithan, J.A. Stankovic (1994), "Scheduling Algorithms and Operating Systems Support for Real-Time Systems," Proceedings of the IEEE, Vol. 82, nº 1, pp. 55-67.