UNIVERSIDADE DE SÃO PAULO
ESCOLA POLITÉCNICA

FRANCISCO RIBACIONKA

Algoritmo Distribuído para Alocação de Múltiplos Recursos em Ambientes Distribuídos

São Paulo
2013

FRANCISCO RIBACIONKA

Algoritmo Distribuído para Alocação de Múltiplos Recursos em Ambientes Distribuídos

Tese apresentada a Escola Politécnica da Universidade de São Paulo para obtenção do título de Doutor em Engenharia Elétrica

Área de Concentração: Sistemas Digitais

Orientadora: Professora Doutora Liria Matsumoto Sato

São Paulo
2013

Este exemplar foi revisado e alterado em relação à versão original, sob responsabilidade única do autor e com anuência do seu orientador.

São Paulo, 6 de agosto de 2013

Assinatura do autor _____

Assinatura do orientador _____

FICHA CATALOGRÁFICA

Ribacionka, Francisco
Algoritmo distribuído para alocação de múltiplos recursos em ambientes distribuídos / F. Ribacionka. -- versão corr. -- São Paulo, 2013.
105 p.

Tese (Doutorado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1.Arquitetura e organização de computadores 2.Programação paralela 3.Algoritmos 4.Computação em nuvem 5.Lógica Fuzzy 6.Processamento de alto desempenho I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II.t.

Dedicatória

Dedico este trabalho em memória de meus pais, Romão Ribacionka e Tereza Derlica Ribacionka que sempre me motivaram a continuar meus estudos.

Agradecimentos

A Deus, por tudo.

A professora Doutora. LIria Matsumoto Sato pela orientação e por ter me aceito como mais um filho em sua família de alunos.

A professora Doutora Luciana Arantes, pela ajuda.

A minha esposa, Márcia Cristina dos Santos Ribacionka, meu filho André dos Santos Ribacionka, minhas irmãs Nanci Ribacionka e Neusa Ribacionka pelo apoio inestimável.

Aos professores doutores Antonio Marcos de Aguirra Massola, Edson dos Santos Moreira, Jaime Simão Sichman, Kechi Hirama e Tereza Cristina melo de Brito Carvalho pela motivação.

Aos professores doutores Hermes Senger e Laécio Carvalho de Barros pela orientação na qualificação deste doutorado.

A professora Maria Cristina Vidal Borba pela valiosa ajuda na língua inglesa.

A todos os colegas do laboratório LAHPC.

A todos os funcionários administrativos da Escola Politécnica da USP.

Aos colegas do CCE: Albina, Camilli, Carlinda, Ettore, Marilda, Marta, Onoe, Rosa Mitie, Sidinei, Stenio e Tatiana, que, mesmo silenciosamente, me ajudaram muito.

Resumo

Ao considerar um sistema distribuído composto por um conjunto de servidores, clientes e recursos, que caracterizam ambientes como grades ou nuvens computacionais, que oferecem um grande número de recursos distribuídos como CPUs ou máquinas virtuais, os quais são utilizados conjuntamente por diferentes tipos de aplicações, tem-se a necessidade de se ter uma solução para alocação destes recursos. O apoio à alocação dos recursos fornecidos por tais ambientes deve satisfazer todas as solicitações de recursos das aplicações, e fornecer respostas afirmativas para alocação eficiente de recursos, fazer justiça na alocação no caso de pedidos simultâneos entre vários clientes de recursos e responder em um tempo finito a requisições. Considerando tal contexto de grande escala em sistemas distribuídos, este trabalho propõe um algoritmo distribuído para alocação de recursos. Este algoritmo explora a Lógica Fuzzy sempre que um servidor está impossibilitado de atender a uma solicitação feita por um cliente, encaminhando esta solicitação a um servidor remoto. O algoritmo utiliza o conceito de relógio lógico para garantir justiça no atendimento das solicitações feitas em todos os servidores que compartilham recursos. Este algoritmo segue o modelo distribuído, onde uma cópia do algoritmo é executada em cada servidor que compartilha recursos para seus clientes, e todos os servidores tomam parte das decisões com relação a alocação destes recursos. A estratégia desenvolvida tem como objetivo minimizar o tempo de resposta na alocação de recursos, funcionando como um balanceamento de carga em um ambiente cliente-servidor com alto índice de solicitações de recursos pelos clientes. A eficiência do algoritmo desenvolvido neste trabalho foi comprovada através da implementação e comparação com outros algoritmos tradicionais, mostrando a possibilidade de utilização de recursos que pertencem a distintos servidores por uma mesma solicitação de recursos, com a garantia de que esta requisição será atendida, e em um tempo finito.

Palavras-chave: Processamento de alto desempenho, Programação paralela, Algoritmo para alocação de recursos distribuídos, Computação em Grade e em nuvem, Lógica Fuzzy.

Abstract

When considering a distributed system composed of a set of servers, clients, and resources that characterize environments like computational grids or clouds that offer a large number of distributed resources such as CPUs or virtual machines, which are used jointly by different types of applications, there is the need to have a solution for allocating these resources. Support the allocation of resources provided by such environments must satisfy all Requests for resources such applications, and provide affirmative answers to the efficient allocation of resources, to do justice in this allocation in the case of simultaneous Requests from multiple clients and answer these resources in a finite time these Requests. Considering such a context of large-scale distributed systems, this paper proposes a distributed algorithm for resource allocation This algorithm exploits fuzzy logic whenever a server is unable to meet a request made by a client, forwarding this request to a remote server. The algorithm uses the concept of logical clock to ensure fairness in meeting the demands made on all servers that share resources. This algorithm follows a distributed model, where a copy of the algorithm runs on each server that shares resources for its clients and all servers take part in decisions regarding allocation of resources. The strategy developed aims to minimize the response time in allocating resources, functioning as a load-balancing in a client-server environment with high resource Requests by customers.

Key-Words: High performance computing, Parallel Programming, Distributed resource allocation algorithm, Grid and Cloud Computing, Fuzzy Logic.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE FÓRMULAS

LISTA DE ABREVIATURAS E SIGLAS

RPM	Requisições Por Minuto
RR	Algoritmo Round Robin
SL	Algoritmo Small Latency
UCP	Unidade Central de Processamento
RMS	Resource Management Service
LRMS	Local Resource Management System
LSF	Load Share Facility
GARA	Globus Architecture for Reservation and allocation
GRAM	Globus Resourc eallocation Manager
QoS	Quality of Service)
API	Application Program Interface
WAN	Wide Area Network
SM	Scheduler Manager

Lista de símbolos

⇔	Congruente, equivalente
→	Aconteceu antes, implicação
∧	E
∨	Ou
∑	Somatório
=	Igualdade

SUMÁRIO

1 INTRODUÇÃO

O uso de sistemas paralelos atualmente está em crescimento. O que era uma tendência, agora é realidade. Os sistemas de computação de alto desempenho, também chamados de HPC (High Performance Computing), são formados por sistemas paralelos com múltiplas CPUs em um único servidor ou por um cluster de computadores ligados por uma rede de interconexão (Breshears, 2009, Parhami, 2002).

Os usuários destes sistemas devem utilizar aplicações paralelas para efetivamente se beneficiarem destas novas tecnologias (Levesque, 2011). Tradicionalmente, as aplicações são escritas de forma sequencial, onde cada instrução é executada uma após a outra. Com as novas tecnologias de processadores com múltiplos núcleos, os programadores devem utilizar técnicas de programação paralela para obterem melhores resultados, e não somente obterem ganhos de desempenho do seu programa com a atualização de seus computadores (Kirk, 2010).

É fato que a atualização do parque instalado ocorre com uma frequência maior que antigamente, devido ao barateamento de preço destes equipamentos. Como consequência natural, as empresas e organizações acabam acumulando equipamentos antigos, que ainda podem apresentar resultados, principalmente se forem agrupados em organizações virtuais, como a computação em nuvem ou a computação em grade (Paula, 2009). Cria-se um grupo de recursos distribuídos, que, através da rede de computadores, podem ser utilizados na execução de aplicações sequenciais ou paralelas.

A tecnologia de grade de computadores não é revolucionária, pois envolve tecnologias existentes como computação distribuída, serviços web, a Internet, tecnologias de criptografia e segurança, além da tecnologia de virtualização (Magoulès, 2009). Cada uma destas tecnologias separadas tem suas aplicações específicas, mas com a tecnologia de grade surge a possibilidade de fornecer um ambiente para a execução de aplicações com maior vazão para processamento.

Com o advento da tecnologia de grade e computação em nuvem, ocorreu um aumento de oferta de acesso a recursos distribuídos, habilitando a comunidade

científica a desenvolver aplicações que utilizam os recursos agregados por esta tecnologia, para resolver problemas científicos de larga escala (Arafah, 2006, Berman, 2003, Krauter, 2002, Foster, 2001). Neste contexto, os seguintes aspectos devem ser atendidos:

- Os recursos são utilizados na execução de aplicações de forma eficiente?
- Na concorrência a estes recursos, há justiça na alocação destes recursos aos pedidos feitos?
- Alguma solicitação fica esperando eternamente por estes recursos?

Neste trabalho, busca-se tratar estes aspectos.

A arquitetura de rede no modelo TCP/IP utilizada atualmente, popularmente conhecida como Internet, possibilitou a criação do modelo de programação cliente-servidor, onde dois sistemas finais trocam informações através de mensagens enviadas por sockets (Kurose, 2009). Uma aplicação de serviços Web típica é formada pelo lado cliente, que solicita uma informação, e o lado servidor, que fornece a informação. Quando um servidor recebe uma grande quantidade de solicitações, uma solução é utilizar um grupo de servidores que procura atender as requisições dos clientes mais rapidamente do que quando há apenas um servidor. Uma alocação apropriada dos servidores para o atendimento das requisições, promovendo um balanceamento de carga nos servidores através de uma distribuição adequada destas solicitações, é necessária para se obter um atendimento mais eficiente (Martinez, 2011, Nakai, 2011).

Reveliotis (2005) apresenta o problema de alocação de recursos distribuídos por uma aplicação cliente-servidor que necessite de recursos formados pela integração de uma coleção de computadores que formam uma plataforma virtual acessível pela Internet, e que o sucesso desta aplicação depende da forma como os recursos que a aplicação necessita para sua execução sejam alocados. A execução de serviços para atender a requisições de um serviço web tem como obstáculo a alocação de recursos para a efetivação deste serviço.

Com a disponibilização de recursos formados por *clusters* de servidores, sistemas *multicores*, estações de trabalho, *desktops*, sistemas de *blade*, servidores para processamento de alto desempenho, ligados por uma rede de interconexão, o usuário tem acesso a uma grande variedade de recursos e que podem ser utilizados

em um processamento distribuído. A alocação destes recursos de forma eficiente e justa é imprescindível.

Há dois modelos para alocação de recursos em sistemas distribuídos: centralizado e distribuído (Leal, 2009). No modelo centralizado, todas as informações e o escalonador de recursos ficam em um nó, que é responsável pela alocação de recursos para todos os nós que a ele estejam ligados e tenham feito solicitações. No modelo distribuído, cada nó que compõe o grupo que compartilha recursos executa uma cópia do escalonador.

O modelo desenvolvido neste trabalho se enquadra nesta segunda categoria. A principal vantagem do modelo distribuído em relação ao centralizado é que não há um único ponto de falha, pois no modelo centralizado, em caso da ocorrência de algum erro no computador onde o escalonador está sendo executado que ocasione a parada deste computador e, consequentemente, a parada da execução do escalonador, todo o sistema fica comprometido.

No trabalho, aqui apresentado, chama-se nó, ou servidor, o sistema com seus recursos próprios, que podem ser computadores com múltiplos núcleos de processamento, como também, *clusters* de computadores, onde vários computadores compõem um grupo formando a imagem de um sistema único com recursos a serem compartilhados, que podem ser utilizados por clientes conectados a estes servidores.

Apresenta-se uma proposta para alocação de recursos distribuídos em uma grade ou nuvem computacional, que provê servidores com múltiplos núcleos de processamento e *clusters* de computadores, proporcionando justiça na alocação destes recursos, ordenando as solicitações enviadas aos servidores de forma a se evitar *deadlock* e *starvation*. É apresentado um algoritmo totalmente distribuído, onde cada nó participa do funcionamento do sistema executando localmente uma cópia do gerenciador de alocação, recebendo e provendo a alocação e execução das requisições. Cada requisição pode requisitar um ou múltiplos recursos.

Os recursos alocados poderão ser de um único “cluster” ou vários servidores e *clusters*, os quais serão selecionados, considerando as suas cargas de processamento, e as distâncias em relação ao nó em que foi solicitada a requisição.

A aplicação deste algoritmo em um sistema que precisa alocar múltiplos recursos distribuídos em um ambiente de grade computacional, conforme já descrito, busca prover as necessidades de desempenho e justiça de atendimento de aplicações sequenciais ou paralelas. Estratégias, que atendam a estes requisitos, não foram encontradas na literatura estudada neste trabalho.

1.1 Objetivos

Considerando um conjunto de sistemas computacionais, como servidores *multicores*, *clusters* de computadores, estações de trabalho, *blade centers*, *desktops* e *notebooks*, cada um com recursos como unidades centrais de processamento, conectados por uma rede de conexão formando um grupo de servidores com o objetivo de compartilhar recursos, acessíveis a clientes conectados a estes servidores, este trabalho tem como objetivo propor e implementar um mecanismo descentralizado de alocação destes recursos distribuídos, garantindo justiça na alocação destes recursos e evitando *deadlock* e *starvation* das requisições.

Esta obra visa oferecer a clientes conectados a estes servidores, a oportunidade de utilizar estes recursos distribuídos de forma eficiente, com qualidade neste atendimento, garantindo o atendimento desta solicitação em um tempo finito. Os recursos utilizados podem, preferencialmente, pertencer ao servidor que recebeu a solicitação, ou pode pertencer a outro nó, ou servidor, do grupo que compartilha recursos.

Neste caso, o algoritmo procura pela melhor opção de servidor para atender esta solicitação. Se vários servidores podem atender a solicitação, utiliza-se de um mecanismo de decisão baseado na lógica fuzzy para esta escolha, que considera a carga de cada nó. O balanceamento de carga impacta no tempo de resposta das aplicações, melhorando o desempenho dos sistemas distribuídos (Srivasta, 2011, Sharma, 2011, Nakai, 2011). Na alocação dos recursos é também considerado o custo de comunicação decorrente das distâncias entre eles e o nó em que foi solicitada a requisição e a potência do processador de cada servidor, reduzindo o tempo de resposta.

Por outro lado, se nenhum servidor pode atender integralmente a solicitação, o algoritmo procura por servidores que, em conjunto, possam atender a solicitação, se houver recursos distribuídos disponíveis em número suficiente para este atendimento.

Este trabalho pode ser aplicado em sistemas paralelos e distribuídos com o objetivo de minimizar o tempo de resposta para as solicitações destes recursos. Tem-se como possíveis utilizações deste trabalho:

- Requisição e alocação de CPUs para processamento paralelo.
- Atendimento de requisições HTTP.

Para atingir os objetivos deste trabalho, foram estabelecidas as seguintes metas:

- Desenvolver um algoritmo que explore uma melhor utilização de recursos que estão distribuídos por *clusters* ligados por uma rede de computadores. Este algoritmo tem como característica ser descentralizado, e deve ser executado em cada servidor que compartilha recursos.
- Implementar neste algoritmo um mecanismo de escolha aos melhores candidatos a ceder recursos.
- Fazer com que este algoritmo não permita a ocorrência de *starvation* e *deadlock.*
- Implementar o algoritmo apresentado e realizar testes para a verificação de seu funcionamento e análise de desempenho, utilizando o simulador de sistemas de grade SIMGrid (Casanova, 2008). Para uma implementação em um ambiente real é necessária a alteração das chamadas de funções de comunicação e de criação de *threads*, pois o SimGrid utiliza funções próprias.

1.2 Motivação

A experiência de utilização de sistemas paralelos no atendimento da demanda de processamento mostra que os recursos não são utilizados de forma integral e eficientemente.

Aplicativos baseados em serviços Web distribuídos também podem ter melhores desempenhos através do uso de mecanismos mais eficientes para alocação dos recursos.

A computação científica é largamente utilizada em várias áreas de pesquisa, como a computação molecular, ciência dos materiais, biofísica computacional, previsão do tempo, agricultura e controle de pragas, pesquisa na área de petróleo e gás, meio ambiente, biologia e medicina, entre outras. A computação de alto desempenho viabiliza a obtenção de resultados, diminuindo o tempo de resposta destas pesquisas.

O investimento em centros de computação científica cresce a cada ano, como pode se verificar no endereço http://www.top500.org. Institutos como o Instituto Avançado de Computação Científica de Riken[1] fomentam a utilização da tecnologia para o desenvolvimento destas áreas de pesquisa.

Esta iniciativa também está na Universidade de São Paulo, com a instalação de um Supercomputador com 2.304 processadores para pesquisa na área de Astrofísica[2] e dos Supercomputadores instalados no Laboratório de Computação Científica Avançada da USP[3]. Estas são iniciativas para fornecer aos pesquisadores o ambiente para processamento científico de suas pesquisas.

Atualmente, escalonadores como o Condor (Thain, 2005) são amplamente utilizados em *clusters* de computadores para atribuir recursos para a execução de aplicações. As pesquisas em andamento para grades computacionais são voltadas a escalonamento de tarefas sequenciais, *workflows* de tarefas e também relacionadas ao balanceamento de carga para servidores web. Até onde foi estudado na literatura, não foi encontrado um modelo desenvolvido como o apresentado neste trabalho, sendo este um dos motivos para a execução desta pesquisa.

Os *clusters* e as grades computacionais estão formando um novo paradigma de utilização em nuvem computacional, abrindo espaço para novas aplicações com novos modelos de serviço. Esta área de atuação também tem muitas possibilidades de investigação e pesquisa para utilizar melhor estes recursos distribuídos.

[1] http://www.aics.riken.jp/en/
[2] http://www5.usp.br/7749/iag-adquire-supercomputador-e-inaugura-laboratorio-de-astroinformatica
[3] http://www.usp.br/lcca

1.4 Estrutura do texto

Este trabalho está estruturado da seguinte forma: no Capítulo 1, está a introdução, o objetivo do trabalho e a metodologia seguida. No Capítulo 2, tem-se a definição dos conceitos referentes ao tema tratado. No Capítulo 3 estuda-se o estado da arte e trata-se o que está em voga nesta área. No Capítulo 4 apresenta-se uma proposta e estratégia para escalonamento de recursos distribuídos. A implementação do algoritmo, os testes e os resultados obtidos são mostrados no Capítulo 5. No Capítulo 6 são apresentadas as contribuições e a conclusão desta tese. As referências bibliográficas vem após o Capítulo 6. Na última parte tem-se um Apêndice com informações sobre a execução da implementação no simulador SimGrid.

2 Conceitos

Neste capítulo estuda-se os conceitos utilizados neste trabalho: Relógio Lógico, Controlador Fuzzy, Starvation e Deadlock, Sistemas Distribuídos e Grade Computacional.

2.1 Relógio Lógico

O conceito de relógio lógico foi descrito por Lamport (Lamport, 1978) e tem como objetivo sincronizar os eventos que ocorrem simultaneamente em um ambiente paralelo e distribuído. Provê a noção de tempo, onde cada evento ocorre antes ou depois do outro.

É de senso comum que um evento *a* acontece antes de um evento *b* se *a* ocorreu em um tempo anterior a *b*. O conceito de relógio lógico pode ser utilizado para a definição desta noção de tempo sem a utilização de um relógio real.

Para esta definição, Lamport assume que o sistema seja composto por uma coleção de processos, cada um deles formado por uma sequência de eventos totalmente ordenados.

A definição de "aconteceu antes" é denotada pelo símbolo "→". Se a e b são eventos do mesmo processo, e a ocorre antes de b, então a→b. Se a→b e b→c então a→c.

A funcionalidade do relógio lógico se dá da seguinte forma: Seja um conjunto *M* de servidores, onde cada servidor é numerado em sequência 1, 2, ..., i, ..., j, ..., n, o relógio lógico do servidor h_i tem prioridade sobre o servidor h_j se h_i for menor do que h_j. Se h_i é igual a h_j, então h_i terá prioridade se i for menor do que j. A ordem de entrada de cada servidor no grupo que vai compartilhar recursos é a numeração sequencial 1, 2, ..., i, ..., j, ..., n do conjunto M.

Neste trabalho define-se como relógio lógico formado pela tupla $<i,relogio_i>$ onde *i* é o valor sequencial do servidor no conjunto M e $relogio_i$ é um valor inteiro, com valor

inicial 0, e é incrementado por cada processo no recebimento de uma mensagem para atualização de dados entre os servidores.

A atualização do relógio lógico local se dá quando um servidor envia uma mensagem para outro. Nesta mensagem é incluída o valor de seu relógio lógico. O servidor que recebe esta mensagem compara este valor com o valor local de seu relógio lógico. Caso tenha um valor inferior ao recebido, ele iguala o seu relógio lógico com o valor recebido na mensagem e o incrementa.

O relógio lógico é usado para ordenar a fila local em cada servidor e auxiliar na tomada de decisão sobre qual servidor vai receber a solicitação. Também tem como objetivo evitar que uma solicitação jamais seja atendida, evitando uma situação de *Starvation*. A ordem se dá na comparação da tupla <*i*, *relógio*$_i$> com <*j, relógio*$_j$>. <i*, relógio*$_i$> é menor do que <*j, relógio*$_j$> se *relógio*$_i$ for menor do que *relógio*$_j$. Caso *relógio*$_i$ for igual a *relógio*$_j$, <*i, relógio*$_i$> é menor do que <*j, relógio*$_j$> se *i*<*j*.

2.2 Controlador Fuzzy

A lógica fuzzy é um tema estudado e aplicado em várias áreas de processamento distribuído e científico (Barros, 2006), (Celikyilmaz 2009), (Mileff, 2006), (Nguyen, 2000), (Park, 1995).

A Lógica Fuzzy foi desenvolvida por Lofti Asker Zadeh em 1965 (Zadeh, 1965) e é utilizada para modelar problemas do mundo real que envolvem a incerteza (Barros, 2006, Klir, 1995) . Os conceitos Fuzzy são modelados como conjuntos fuzzy, que é uma generalização da teoria de conjuntos tradicional.

Um controlador fuzzy é um método de tomada de decisão baseado no raciocínio aproximado através de regras proposicionais fuzzy (Ribacionka, 2008). As regras proposicionais fuzzy tem a forma Se *proposição fuzzy* então *ação fuzzy*, onde a *proposição fuzzy* é formada por variáveis observadas do processo controlado e *ação fuzzy* é a ação a ser seguida.

Neste trabalho, as variáveis observadas para formar as proposições fuzzy são a latência entre os servidores e a potência de processamento de cada servidor. Quando um servidor recebe uma requisição de recursos e não tem recursos próprios

para atender esta requisição, o controlador fuzzy é utilizado quando mais de um servidor, que participa do grupo de servidores que compartilham recursos, pode atender a requisição. A *ação fuzzy* ajuda a decidir qual é a melhor opção de escolha para qual servidor enviar a solicitação de recursos.

A Figura 1 mostra o esquema de um controlador fuzzy.

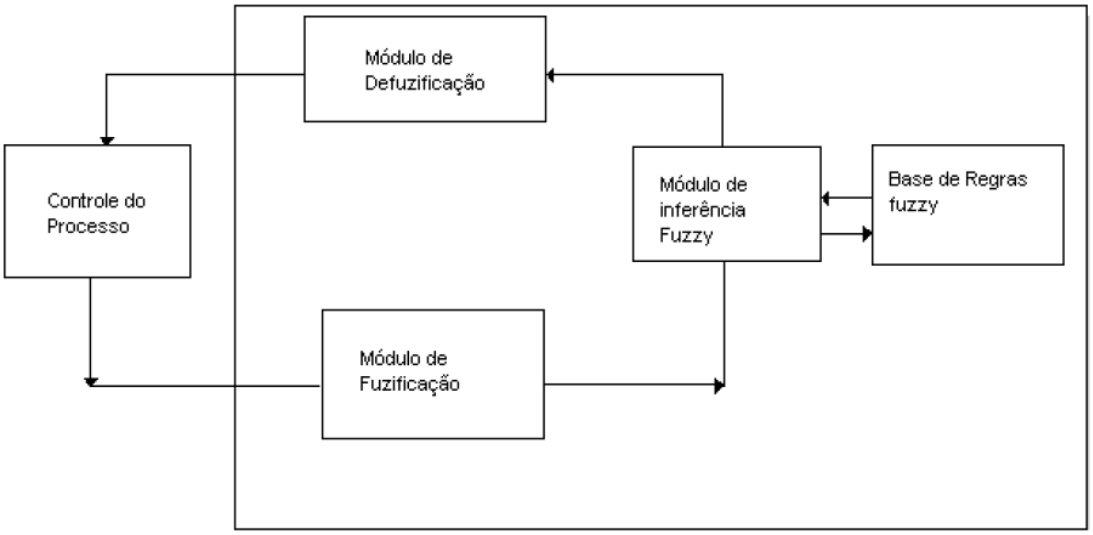

Figura 1 - Controlador Fuzzy (Ribacionka, 2008)

Um controlador fuzzy é composto por quatro módulos. O módulo de fuzificação é o módulo onde os valores de entrada do sistema são convertidos para conjuntos fuzzy, com as respectivas faixas de valores onde estão definidos. O módulo de inferência precisa do módulo da base de regras fuzzy. Nesta base são definidas as proposições fuzzy que serão avaliadas de acordo com as variáveis de entrada para compor a variável de saída. O módulo de inferência fuzzy valora estas proposições fuzzy e as combina, para produzir a saída. O módulo de defuzificação é o processo de conversão do conjunto fuzzy de saída em um número que melhor represente este conjunto.

Na modelagem de um controlador fuzzy, a primeira tarefa é definir quais informações fluem pelo sistema, quais transformações são executadas nestas informações e quais são as informações fornecidas pelo sistema. Neste trabalho foi identificada como variáveis de entrada, a potência de processamento e a latência de rede entre os servidores.

Na próxima etapa do projeto de desenvolvimento de um controlador fuzzy, define-se os conjuntos fuzzy para cada variável que compõe o sistema. O particionamento de cada variável em subconjuntos fuzzy define os seus respectivos domínios.

Cox (1999) apresenta um exemplo de conjunto fuzzy que modela o Custo de um Projeto. Este exemplo está na Tabela 1.

Tabela 1 - Conjunto Fuzzy de entrada

Custo de Projeto			
Subconjunto	*Início*	*Fim*	*Abreviação*
Normal	0	400	NO
Moderado	250	750	MO
Alto	600	900	AL
Aceitação			
Subconjunto	*Início*	*Fim*	*Abreviação*
Baixa	0	4	BA
Média	3	7	ME
Alta	6	10	AL

As proposições condicionais fuzzy descrevem o conhecimento sobre o sistema modelado, e tem a forma: se *proposição fuzzy* então *ação fuzzy*. No exemplo apresentado na Tabela anterior pode-se construir as seguintes regras:

- Se Custo de Projeto é Normal então Aceitação é Alta
- Se Custo de Projeto é Moderado então Aceitação é Média

Estas proposições formam a base de regras.

A defuzificação é a sequência de operações para transformar a saída fuzzy em um número real que melhor o represente (Klir, 1995, Nguyen, 2000). Segundo Barros (2006), "qualquer número real, que de alguma maneira possa representar razoavelmente o conjunto fuzzy B pode ser chamado de um defuzificador de B". O valor obtido pela defuzificação mostra a ação a ser tomada pelo controlador para obter a resposta solicitada.

2.3 *Deadlock* e *Starvation*

Deadlock pode ser definido formalmente como segue: Um conjunto de processos está no estado de *deadlock* se cada processo do conjunto está esperando por um evento que apenas outro processo do conjunto pode causar (Tanenbaum, 2010). Por exemplo, em um sistema com dois processos, A e B. Ocorre *deadlock* se o processo A está esperando por um evento que só pode ser causado pelo processo B e o processo B está esperando por um evento que só pode ser causado pelo processo A.

Silberschatz (2005) mostra quatro condições necessárias, que devem ocorrer simultaneamente, para a ocorrência de *deadlock*:

1. Ao menos um recurso deve ser mantido em um modo não compartilhado, ou seja, somente um processo por vez pode utilizar o recurso. Se outro processo requisita o recurso, esta requisição deve ser adiada até a liberação do recurso.
2. Um processo deve obter pelo menos um recurso e esperar para adquirir outros recursos adicionais que estão sendo utilizados por outros processos.
3. Um recurso em utilização só pode ser liberado após a conclusão do processo que o utiliza.
4. Deve haver uma fila circular de dois ou mais processos, cada um dos quais está à espera de um recurso ocupado por um dos processos desta fila.

Silberschatz (2005) afirma que se uma destas quatro condições não for satisfeita, não ocorrerá *deadlock*.

Outro problema relacionado com *deadlock* é o bloqueio infinito ou *starvation*. Em um sistema dinâmico onde os processos fazem requisições de recursos, uma política de decisão para qual processo deve ser alocado o recurso deve ser definido, de forma que nenhum processo, mesmo não estando no estado de *deadlock*, não fique esperando por um tempo infinito até conseguir o recurso solicitado (Tanenbaum, 2010).

2.4 Sistemas Distribuídos

Um sistema distribuído é composto por vários computadores conectados por canais de comunicação, capacitando-os a enviar mensagens entre eles (Ben-Ari, 2006).

Colouris em (Coulouris, 2001) define sistema distribuído como um sistema onde seus componentes, localizados em uma rede de computadores, se comunicam e são coordenados somente por passagem de mensagens, e com as seguintes características:

- Concorrência de componentes: são permitidos a execução simultânea de programas e o compartilhamento de recursos entre seus usuários. O sistema tem capacidade de manipular este compartilhamento e de incluir novos recursos compartilhados.
- Falta de um relógio global: significa que ações coordenadas entre programas são realizadas através de trocas de mensagens para sincronização.
- Falha de seus componentes: os componentes do sistema distribuído podem falhar isoladamente, sem gerar a parada de todo sistema distribuído.

Recursos podem ser gerenciados por servidores e acessados pelos seus clientes.

Os desafios inerentes à construção de sistemas distribuídos são:

- A heterogeneidade de seus componentes: diferentes componentes de hardware e software funcionam coordenadamente.
- A facilidade de se adicionar ou substituir componentes abertos: Componentes desenvolvidos com software livre e com hardware heterogêneo de vários fabricantes.
- Segurança das informações: garantia de confidencialidade (proteção ao acesso não autorizado), integridade destes dados e disponibilidade destes dados aos seus usuários.
- Escalabilidade: habilidade de funcionar bem quando o número de usuários e o número de recursos cresce.
- Manipulação de falhas: recuperação de um estado de programas e dados antes da falha ter ocorrido.

- Concorrência dos seus componentes: capacidade dos serviços e aplicações poderem ser utilizados por vários clientes simultaneamente.
- Transparência em sua utilização: habilidade do sistema se apresentar, ao usuário, como um componente único e uniforme, e não como uma coleção de componentes independentes.

2.5 Grade computacional

Computação em grade tem como objetivo criar um computador virtual composto por uma coleção de sistemas heterogêneos ligados por rede, compartilhando recursos de armazenamento e de processamento (Ferreira, 2003), (Arafah, 2006). É visto como uma infraestrutura que provê recursos para a criação de organizações virtuais (VO) que consiste de recursos geograficamente distribuídos mas que aparenta ter a função de uma única organização com um objetivo único.

Grades proporcionam acesso a recursos geograficamente distribuídos não apenas para aplicações científicas, mas também para aplicações financeiras, administrativas, governamentais e de diversão (Berman, 2003). Um elemento importante de qualquer grade é a rede, que liga os recursos que estão distribuídos para permitir a execução nesta grade de uma determinada aplicação.

Estas aplicações são submetidas e executadas em ambientes de gerenciamento (Thain, 2005), (Holt, 2005) que oferecem mecanismos de alocação de recursos.

Uma grade computacional tem como objetivo fornecer processamento distribuído para computação científica, formado por um conjunto de hardware e de software, que fornece acesso a recursos distribuídos, para os usuários de forma transparente (Paula, 2009, Foster, 2001).

O software de gerenciamento da grade deve organizar recursos e usuários que pertencem a diferentes grupos, cada um com suas políticas próprias de acesso, e utilização destes recursos, de forma a unir estas organizações em uma Organização Virtual (VO), formando uma unidade de acesso. O desafio é garantir que um usuário tenha acesso a estes recursos distribuídos, que, se acessados individualmente, seria necessário mecanismos diferentes de segurança, como login e senha. Mas na

Organização Virtual, um único mecanismo de acesso vai permitir ao usuário utilizar os recursos distribuídos nestas organizações utilizando-se de apenas um login e senha.

O software de grade deve permitir a utilização de protocolos abertos e de uso comum, conhecidos, para garantir o perfeito funcionamento entre os componentes da grade, com qualidade de serviço e segurança, permitindo o acesso e disponibilidade destes recursos.

Outra característica de uma grade de computadores é o de permitir o crescimento por meio da inclusão de novos recursos. Com este crescimento, podem ocorrer erros e falhas durante a utilização da grade, ocorrendo flutuação na disponibilidade de recursos. Para o correto funcionamento da grade, deve haver tolerância a falhas de hardware e de software.

Krauter (2002) classifica grade de computadores em três categorias: grade de processamento, grade de serviços e grade de dados.

No primeiro caso, o foco é permitir o processamento remoto de programas paralelos e distribuídos, qualificando estes recursos como uma plataforma de execução de aplicações. Neste sentido, as estratégias de alocação de recursos tem um alto grau de importância, pois o escalonador de aplicações não gerencia os recursos locais disponíveis. Esta é uma responsabilidade do gerenciador local de recursos, implicando na autorização deste gerenciador para a utilização destes recursos.

No segundo caso, em grade de serviços, se enquadram aplicações cliente-servidor onde, um programa implementa uma interface no sistema final para o usuário ter acesso remoto a um servidor. Esta arquitetura utiliza vários protocolos da camada de aplicação já estabelecidos, como o HTTP ou HTTPS, fornecendo infraestrutura que pode crescer sob demanda do usuário.

Uma grade de dados tem como objetivo fornecer ao usuário um serviço de acesso e uso de base de dados com o crescimento da capacidade de armazenamento, conforme recursos são agregados à grade. Esta é uma área onde há muito a ser feito, pois a crescente demanda pela utilização de sistemas gerenciadores de banco de dados em aplicações comerciais e científicas exigem transparência no acesso aos dados, e em aplicações de banco de dados distribuídos a grade deve fornecer transparência ao usuário.

3 Estado da arte

Este Capítulo descreve os modelos de escalonamento de tarefas e as pesquisas relacionadas a este trabalho.

Se for considerado que há apenas um recurso para ser compartilhado, o problema de alocação de recursos distribuídos é equivalente ao problema de exclusão mútua distribuída.

Algoritmos de exclusão mútua podem ser divididos em duas classes: baseado em permissão (Lamport, 1978), (Ricart, 1981), (Maekawa,1985) e baseado em token (Suzuki, 1985), (Naimi, 1996).

Os algoritmos do primeiro grupo são baseados no princípio que um nó entra em sua sessão crítica somente após receber a permissão de todos os outros nós, ou a maioria deles (Raynal, 1991). No segundo grupo, um único token é compartilhado entre todos os nós, e a sua posse permite ao nó a exclusividade de utilizar a região crítica.

O problema de exclusão mútua-k (k-mutex) é uma generalização do problema de exclusão mútua considerando k unidades que acessam estas unidades simultaneamente, ou um processo por unidade. Portanto, um algoritmo k-mutex deve garantir que pelo menos k processos podem utilizar sua sessão crítica ao mesmo tempo. Muitos algoritmos de exclusão mútua-k são propostos na literatura (Srimani, 1992), (Reddy, 2008).

Raynal (1991) e Baldoni (1994) estenderam o problema de exclusão mútua-k para o problema de alocação distribuída k de M: um processo solicita por k recursos pertencentes a M nós. Entretanto, contrariamente a este trabalho, nenhum destes dois trabalhos controlam qual recurso é alocado para determinado processo. Eles apenas garantem que um processo terá exclusividade para usar um recurso.

3.1 Modelos de escalonamento

Nos artigos estudados verificou-se que basicamente há dois modelos para escalonamento de tarefas em grades computacionais: centralizado e distribuído. No modelo centralizado, todas as informações ficam em um servidor, que é responsável pela distribuição de tarefas para todos os servidores que a ele estejam ligados. No modelo distribuído, cada servidor que compõe a grade possui o seu sistema de escalonamento de tarefas, e a forma da tomada de decisão de quem vai executar a tarefa varia de acordo com o algoritmo desenvolvido. A Figura 2 ilustra um modelo genérico descentralizado:

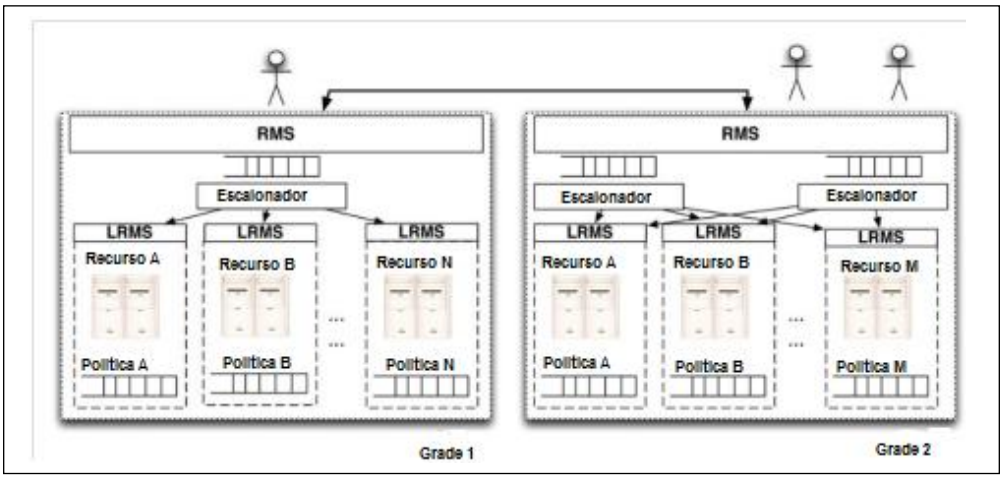

Figura 2 - Modelo descentralizado. Adaptado de (Leal, 2009)

Na Figura 2 tem-se os seguintes componentes:

RMS: Resource Management Service. É o componente da grade que provê uma interface ao usuário para requisição de recursos da grade.

LRMS: Local Resource Management System. Escalonador local (exemplos: Condor, Load Sharing Facility, Sun Grid Engine).

Park (1995) apresenta uma abordagem que caracteriza o estado inerente de incerteza global em um grande sistema distribuído em termos da teoria dos conjuntos fuzzy e apresenta um algoritmo de balanceamento de carga distribuída que, explicitamente, reflete o efeito da incerteza no processo de tomada de decisão. A noção de variáveis linguísticas é usada para variáveis de entrada do modelo, que têm valores de estado imprecisos e incertos. Um controlador fuzzy foi desenvolvido e permite que cada nó tome decisões flexíveis, de acordo com as entradas fuzzy.

Resultados obtidos em simulações mostram que o algoritmo proposto produz um desempenho superior, reduz substancialmente o número de mensagens requeridas e, geralmente, transfere menos tarefas, em comparação com outros dois algoritmos de balanceamento de carga distribuídos.

Weissman (1996) apresenta um modelo de escalonamento em uma rede WAN (Wide Area Network) é descrito. A rede WAN é formada por uma coleção de servidores. Cada um deles é um domínio administrativo com suas próprias políticas de segurança, sistema de arquivos, procedimentos de contabilidade do sistema, além das políticas de utilização de seus recursos computacionais. Cada servidor executa um gerenciador de escalonamento (SM) tais como o Condor e LoadLeveler, entre outros. O modelo descrito de escalonamento WAN é um algoritmo distribuído composto por duas partes. O componente SM local do servidor, e o componente remoto, que formam uma hierarquia, onde o SM remoto consulta os SMs locais para obter candidatos a receberem tarefas.

No artigo (Foster, 1999) os autores descrevem o GARA (Globus Architecture for Reservation and Allocation), uma extensão do GRAM (Globus Resource Allocation Manager). O objetivo do GARA é garantir o QoS (Quality of Service) de aplicações distribuídas, com atuação em quatro áreas: descoberta dinâmica de recursos, reserva destes recursos, heterogeneidade de implementações e independência administrativa. GARA introduz o termo genérico para objeto como recurso, que pode ser um dos seguintes recursos: fluxo de rede, blocos de memória, blocos de disco e entidades de processamento, e introduz também o conceito de reserva destes recursos. É um conjunto de *Application Program Interface* (API) que permitem a submissão, monitoramento e término de tarefas.

Subramani (2002) descreve o funcionamento dos modelos de escalonamento de tarefas centralizado, hierárquico e distribuído e propõe uma melhora no modelo distribuído.

No modelo centralizado, o escalonador principal mantém informações sobre todos os servidores, e todas as tarefas são submetidas para este escalonador, que toma todas as decisões sobre escalonamento. Neste modelo, o servidor local não toma nenhuma decisão, apenas encaminha a tarefa ao escalonador principal, informando-o quando tarefas encerram e recursos computacionais são liberados. Subramani

(2002) afirma que o modelo centralizado não é totalmente escalável porque o escalonador principal precisa manter muitas informações detalhadas sobre todos os servidores, além de que no modelo centralizado não é simples o uso de diferentes esquemas de prioridade em diferentes servidores. Esta adversidade afeta tarefas locais, beneficiando tarefas remotas.

No modelo hierárquico, o processo de escalonamento é compartilhado entre o escalonador principal e os servidores locais. A diferença com o modelo anterior é que a tarefa não é mantida em uma fila de submissão no escalonamento principal. No momento em que a tarefa chega, o escalonador principal encaminha esta tarefa ao servidor com a melhor perspectiva de menor tempo de execução para esta tarefa, e a tarefa então é enfileirada no escalonador local deste servidor.

No modelo distribuído, há um meta-escalonador em todos os servidores. Este meta-escalonador troca informações com todos os outros servidores. Se algum destes tem um baixo nível de processamento, a tarefa é transferida para ele. Como uma tarefa local sempre é submetida para a sua fila local, o esquema distribuído é mais escalável do que o modelo hierárquico.

O modelo proposto por Subramani (2002) é uma melhoria do modelo hierárquico, onde a tarefa local também é enviada a todos os servidores com menor fila de processamento. Esta tarefa fica enfileirada em todos eles, e quando um servidor tem processamento livre e pode iniciar a execução desta tarefa, um aviso é emitido a todos os servidores restantes, e o processamento desta tarefa então pode começar.

No artigo (Schopf, 2004) o autor apresenta ações que um escalonador de alto desempenho em uma grade computacional deve seguir para fornecer benefícios para as aplicações, em três fases: descoberta de recursos, onde é gerada uma lista de recursos disponíveis; levantamento de informações para a tomada de decisão e a execução da tarefa. Seguindo as ações descritas, o autor afirma que as novas gerações de escalonadores para aplicações em grade serão eficientes e trarão qualidade na execução dos programas.

Milef (2006) concentra-se em uma nova geração de técnicas adaptativas de balanceamento de carga dinâmico, que é baseada em tecnologia J2EE[4] e pode ser

[4] Java2 Platform Entreprise Edition (J2EE), tecnologia que possibilita o desenvolvimento de aplicações Java (http://www.oracle.com/technetwork/java/javaee/overview/index.html)

aplicada em servidores de aplicação desta plataforma. Este artigo discute em detalhe o modelo teórico de balanceamento de carga e sua realização prática para sistemas de comércio eletrônico que simultaneamente servem muitos clientes que transmitem um grande número de solicitações. Neste estudo, os autores utilizam a lógica fuzzy como máquina de decisão no balanceamento de carga, com três variáveis linguísticas fuzzy: uma para o valor dos dados de entrada e saída, outro para a utilização da CPU e um terceiro que indica a capacidade de serviço de um nó do servidor.

Leal (2009) apresenta o conceito de Federação de Grades. Atualmente são três os conceitos de grades: empresarial, grade de parceria e grade utilitário. A grade empresarial é definida por recursos da mesma instituição são compartilhados pelas divisões da própria organização. A grade de parceria é formada pela combinação de recursos de organizações que forma uma parceria para gerar uma infraestrutura de computação, como o TeraGrid Americano ou EGEE Europeu e Grade utilitário, para prestação de serviços a terceiros. A Federação de Grades permite a união destes tipos diferentes de grades.

No mesmo artigo, Leal (2009) apresenta um modelo descentralizado para o escalonador de tarefas independentes em Federações de Grades. Este modelo consiste em um conjunto de meta-escalonadores em cada uma das grades pertencentes à Federação. Um meta-escalonador no topo da hierarquia desta federação possui informações genéricas sobre a Federação, e quatro algoritmos são apresentados com o objetivo de maximizar a utilização dos recursos da Federação: objetivo estático, objetivo dinâmico, objetivo estático com escalonamento avançado e objetivo dinâmico com escalonamento avançado. Estes quatro algoritmos são baseados em um modelo de performance da estrutura que forma a Federação de Grades, não o seu estado.

Paula (2009) propõe e implementa o sistema centralizado GCSE (Grid Cooperative Scheduling Environment) que provê uma estratégia de escalonamento cooperativo para usar eficientemente os recursos distribuídos por vários *clusters* e computadores, todos conectados a redes de comunicação pública.

Srivasta (2011) descreve uma estratégia dinâmica de balanceamento de carga para aumentar a eficiência e o desempenho na distribuição de tarefas em uma grade

computacional. Neste trabalho, é proposto um algoritmo, que possui quatro etapas: monitorar a performance das estações de trabalho, sincronizar as estações de trabalho através da troca de mensagens com informações de performance, efetuando o cálculo de carga de trabalho e tomando as decisões sobre a migração de tarefas. No algoritmo de balanceamento de carga proposto, há quatro tipos de atividades: o recebimento de novas tarefas e o enfileiramento de um tarefa em uma estação particular, a execução completa de uma tarefa, a recepção de novos recursos e a retirada de um determinado recurso.

Martinez (2011) propõe um sistema de balanceamento de carga para computação interativa que capacita o balanceamento de carga para códigos interativos baseados em sistemas multicore Linux que contenham recursos heterogêneos. A abordagem do trabalho mostra uma adaptação da carga computacional dos processadores, balanceando a carga entre os nós heterogêneos, passando a carga de processadores sobrecarregados a outros livres. Os autores afirmam que a utilização eficiente destes recursos pode melhorar significativamente o desempenho de sistemas paralelos.

O trabalho de Sharma (2011) é aplicado a balanceamento de carga a servidores web, onde a distribuição se baseia no número de requisições. O autor propõe uma solução onde os servidores são divididos em *clusters*, e desenvolve um modelo para resolver o problema da distribuição das requisições entre estes *clusters*.

O trabalho feito por Jiang (2011) fornece uma proposta para resolver o problema de alocação de recursos em uma grade ou nuvem computacional. Neste trabalho, uma grade ou nuvem computacional é composta por um conjunto *P* de processos e um conjunto *R* de recursos compartilhados, cada um com tipos diferentes e que podem ser acessados em uma forma de exclusão mútua. Os processos podem se comunicar entre eles através de troca de mensagens, e periodicamente um processo pode requerer para entrar em sua região crítica para acessar alguns destes recursos. Um processo p_i pertencente a *P* entra em sua região crítica para depois receber os recursos solicitados. Após a utilização destes recursos, p_i sai de sua região crítica e libera todos os recursos adquiridos. Os processos são determinados a sair de sua região crítica em um tempo finito. O problema de alocação de recursos se preocupa em como garantir que todos os recursos sejam

acessados através de exclusão mútua, e que todos os processos que precisem acessar a sua região crítica consigam este acesso em um tempo finito. Para isto, o autor propõe a construção de um modelo que pode ser utilizado para gerar uma solução para o problema de alocação de recursos.

Wang (2012) os autores tratam do problema apresentado por um cenário onde a comunidade formada por cientistas que utilizam recursos distribuídos geograficamente, formado por inúmeros nós de processamento e que não são confiáveis, pois alguns nós podem falhar. Estas falhas afetam a utilização destes recursos pelos cientistas. O trabalho apresentado neste artigo propõe um mecanismo para execução de tarefas neste ambiente distribuído, baseado no modelo de confiança no relacionamento social das pessoas. Uma relação de confiança é criada entre os nós de processamento e a confiabilidade no sistema é avaliada através do método cognitivo Bayesiano. Os autores propõe um algoritmo que pode ser utilizado para aumentar a confiança na utilização destes recursos distribuídos em processamento de tarefas distribuídas em larga escala, de forma segura.

Jing (2012) apresenta um algoritmo para escalonamento de máquinas virtuais em servidores físicos em um ambiente de computação em nuvem. O objetivo deste algoritmo é de maximizar os benefícios dos provedores de serviços no caso dos recursos não serem suficientes para atender a todas as requisições de recursos nesta nuvem computacional. Cada requisição solicita uma ou mais máquinas virtuais, informando na requisição o tempo de utilização prevista para cada máquina virtual. Esta informação é utilizada pelo algoritmo para determinar, pelo módulo escalonador do algoritmo, qual é o melhor servidor que será escolhido para dar suporte a criação da máquina virtual.

Beumont (2013) considera o problema de encontrar recursos em servidores no modelo Cliente-Servidor, onde os servidores possuem graus de capacidade, caracterizados pela quantidade de operações de ponto flutuante (FLOPS) que o servidor pode processar em um ciclo de processamento e o número máximo de conexões TCP que o servidor pode manter abertas simultaneamente. O objetivo é de encontrar recursos onde o número de clientes atribuídos a um servidor é menor do que um valor determinado, e a soma de todas as requisições não ultrapassa a

capacidade total do servidor. Considerando servidores representados por máquinas físicas e os clientes representando serviços, que podem ser desenvolvidos nos servidores através de uma ou mais máquinas virtuais. Cada serviço gera uma demanda e uma máquina física pode hospedar pelo menos uma máquina virtual.

Khazaei (2013) propõe um modelo de análise de desempenho na alocação de recursos em centros de computação de nuvem. Neste artigo, define-se que um centro de computação de nuvem é composto por um número de máquinas físicas, e que são alocadas aos usuários conforme uma ordem no recebimento das requisições de recursos feitas pelos usuários. Um usuário de nuvem computacional pode solicitar mais de uma máquina virtual em uma única requisição. O modelo proposto assume o recebimento das requisições dos usuários segundo a distribuição de Poisson, suporta alto grau de virtualização (mais de dez máquinas virtuais em cada máquina física) e trata diferentes atrasos impostos pelos centros de computação de nuvem e as requisições dos usuários.

A pesquisa em algoritmos de submissão de tarefas é outra área onde há vários trabalhos publicados. A proposta de um algoritmo genético para submissão de tarefas que pode ser aplicado em um ambiente de grade que tenha tolerância a falhas para entrada de tarefas é feita por Chao-Chin (Chao-Chin, 2010), migração de tarefas com ou sem *checkpoint* e mecanismos de replicação de tarefas.

Zong (2012) apresenta uma proposta de algoritmo para escalonamento de tarefas em computação em nuvem híbridas. Os autores afirmam que existem quatro modelos de computação em nuvem: públicas, privadas, comunitárias e híbridas. A proposta deste trabalho é o de apresentar um recurso para submissão de tarefas em uma nuvem computacional formada por nuvens privadas e nuvens públicas, e o algoritmo apresentado otimiza os custos operacionais da utilização destes recursos.

Chen (2012) apresenta um algoritmo de escalonamento de tarefas para melhorar a eficiência na utilização de energia por computadores de alto desempenho, fazendo com que alguns nós selecionados passem para o estado ocioso, caso alcancem um estado com baixo valor de utilização. O algoritmo proposto armazena os arquivos com informações do sistema sobre os estados do sistema com relação ao consumo de energia, alterando, dinamicamente, o estado dos nós selecionados para ocioso.

Em Ting (2012) os autores consideram o problema de escalonamento de tarefas com prioridade baixa em recursos computacionais com baixa carga de utilização, em uma nuvem computacional. O escalonador proposto coloca as tarefas com baixa prioridade em um estado de suspensão, quando tarefas com maior prioridade solicitem recursos para a nuvem, evitando assim, conflitos na execução destas tarefas.

O estudo de algoritmos de balanceamento de carga tem sido feito em vários trabalhos, e é um tópico de pesquisa fundamental em computação paralela devido ao uso eficiente de múltiplos recursos heterogêneos e que pode melhorar a performance do sistema (Srivasta, 2011) (Park, 1995) (Martínez, 2011) (Bahi, 2010) (Sharma, 2011) (Mileff, 2006).

Nakai (2011) descreve um algoritmo de balanceamento de carga que reduz o tempo de resposta para servidores web distribuídos geograficamente, redirecionando pedidos excedentes para outro servidor, sem sobrecarregá-lo. Os autores afirmam que o tempo de resposta de um pedido redirecionado por um servidor web é afetado por dois fatores: o tempo que o servidor web remoto leva para processar o pedido e a latência entre os servidores web. É proposto um algoritmo que permite que o compartilhamento de recursos entre os servidores web, onde cada servidor web pode assumir dois estados diferentes: a de fornecedor e de consumidor. Os fornecedores podem compartilhar recursos com outros servidores web, e os consumidores são servidores web sobrecarregados que consomem recursos compartilhados pelos fornecedores. Este algoritmo propõe resolver o problema de escolher os servidores web remotos, a fim de redirecionar as solicitações, e que tenta manter a fila de solicitações locais menor do que um limite, sem sobrecarregar os servidores web remotos.

Nakai (2011) apresenta um exemplo de uma aplicação web com seis servidores, réplicas entre si, distribuídos pelo mundo, como servidores PlanetLab (HTTP://www.planet-lab.org) no Brasil (S1), Estados Unidos (N1), Bélgica (E1), Áustria (E2), Japão (A1) e China (A2), conforme observado na Figura 3:

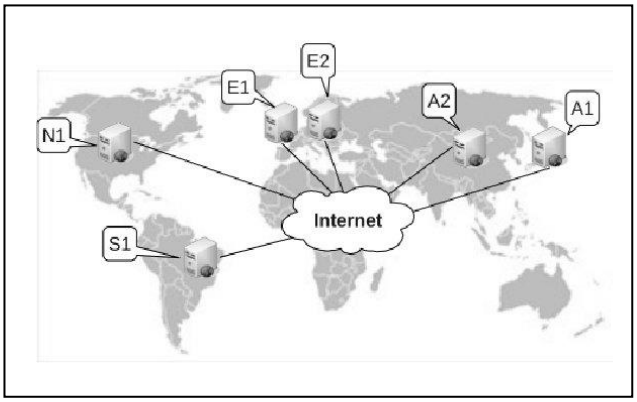

Figura 3 - Réplicas de servidores web (Nakai, 2011)

A Tabela 2 apresenta a latência, em milissegundos, entre estes servidores.

Tabela 2 - Latência entre os servidores (Nakai, 2011)

	S1	N1	E1	E2	A1
N1	89				
E1	138	48			
E2	140	58	18		
A1	193	109	151	162	
A2	272	156	114	122	68

Neste cenário com seis réplicas do mesmo servidor web, é estabelecido um limite de 100 requisições por segundo. Qualquer valor acima deste limite é considerado uma sobrecarga.

A lógica fuzzy também é utilizada em algoritmos de alocação de recursos. Smith (2000) propõe um algoritmo baseado em lógica fuzzy para alocação de recursos distribuídos para aplicações em tempo real. O conjunto de recursos é composto por recursos militares como navios e aviões, cada um equipado com sensores de comunicação e radar. Neste algoritmo, o controlador fuzzy toma decisões baseados em regras da lógica fuzzy.

Uma arquitetura de um sistema multi-agente que propõe uma metodologia de aprendizado de máquina é apresentado por Khuen (Khuen, 2011). Um algoritmo fuzzy é utilizado para auxiliar no aprendizado do comportamento dos agentes no processo de negociação adequada para um problema de alocação de recursos. Esta

arquitetura é dividida em três partes: módulo de aprendizagem, módulo de inferência Fuzzy e o módulo com as funções de pertinência e com a base de regras fuzzy.

As principais diferenças entre os trabalhos (Smith, 2000) e (Khuen, 2011) com este trabalho são, primeiro, que ambos tem o módulo de decisão centralizada, enquanto a solução proposta por esta tese é distribuída e segundo, que o algoritmo desenvolvido nesta tese pode ser aplicado em computação em grade com diferentes configurações, diferentemente destes dois trabalhos que são basicamente aplicações específicas da lógica fuzzy na alocação de recursos distribuídos.

3.2 Algoritmos distribuídos baseados em permissão

Raynal (1991) propôs um algoritmo que pertence à família de algoritmos distribuídos baseados em permissão, que considera, em um sistema distribuído, um conjunto de recursos idênticos que podem ser manipulados por requisições de múltiplos processos, e que implementa um mecanismo para alocação destes recursos para estes processos.

Este trabalho considera, em um sistema distribuído, um conjunto de *M* recursos idênticos compartilhados por *n* processos. Cada recurso pode ser utilizado por pelo menos um processo em um dado momento. Um processo *P* pode solicitar *k* recursos e este processo fica bloqueado até conseguir estes recursos. Ao conseguir estes recursos, o processo somente pode fazer outra solicitação após liberar os recursos que está utilizando.

3.3 Considerações sobre os modelos de algoritmos

Observa-se na literatura estudada ((Subramani, 2002) (Leal, 2009) (Srivasta, 2011) (Nakai, 2011)) que as soluções propostas tratam do escalonamento de tarefas entre servidores de recursos, procurando um balanceamento de carga na execução destas tarefas.

Raynal (1991) apresenta um algoritmo de alocação de recursos, onde processos, em execução em diversos servidores solicitam múltiplos recursos. Em sua proposta os recursos estão em um único servidor e garante que não ocorre *deadlock* e *starvation*.

Não foi encontrada na literatura estudada uma proposta de alocação de recursos distribuídos, onde cada servidor com recursos possibilite a alocação, por um cliente, de recursos pertencentes a vários servidores que somados atendam a requisição.

As grades computacionais atuais, compostas por vários domínios, com sistemas com múltiplos *clusters* agrupados, geram a necessidade de uma solução onde permita a alocação de recursos distribuídos por vários servidores, e as soluções apresentadas na literatura mostram-se insuficientes para utilizar eficientemente os recursos disponibilizados por estas grades computacionais.

4 Algoritmo descentralizado para alocação dinâmica de recursos distribuídos

Neste Capítulo é apresentado o algoritmo para alocação de recursos distribuídos, que pode ser aplicado em uma rede de computadores e que utiliza a lógica fuzzy para tomada de decisão. É um modelo descentralizado, onde uma cópia do algoritmo é executada em cada membro que compartilha recursos.

Neste modelo, cada servidor recebe requisições como também compartilha os seus recursos. Uma requisição pode solicitar um ou mais recursos a um servidor e é prioritariamente atendida por ele próprio, caso tenha recursos livres suficientes para o seu atendimento. Caso contrário, é encaminhada uma alocação em outros servidores.

Considera-se um conjunto de servidores S_1, S_2, … S_k. Cada servidor S_i possui m_i recursos e tem um cliente conectado a ele enviando solicitações de recursos. Os servidores alocam recursos locais ou remotos e executam estas requisições.

Identificadores únicos são atribuídos aos servidores, clientes e recursos, como s_i, c_j res_k identificam o servidor *i*, o cliente *j* e o recurso *k*, respectivamente.

A estratégia adotada busca minimizar o tempo de resposta das requisições. Para tanto, a priorização de requisições a um servidor de serem atendidas localmente contribui para esta redução. Na alocação em outros servidores que tenham recursos livres leva-se em consideração parâmetros como:

- a latência de comunicação entre o servidor de entrada da requisição como também entre os servidores a serem selecionados para o atendimento, buscando minimizar o custo com comunicação;
- a capacidade de processamento dos servidores que podem ser heterogêneos;
- o número de recursos que se encontram livres em cada servidor.

A combinação destes critérios é feita por um mecanismo baseado em lógica fuzzy que fornece um valor com o qual se toma a decisão sobre a seleção dos servidores. Tal componente consulta uma estrutura global denotada Mapa de Disponibilidade,

que mantém informações atualizadas sobre a disponibilidade de recursos. Explorando a lógica fuzzy o algoritmo procura assegurar o melhor tempo de resposta e o balanceamento de carga dos recursos utilizados.

O Mapa de Disponibilidade é uma estrutura usada pelo algoritmo para verificar quais servidores são elegíveis para receber requisições remotas. Nele é mantido o número de recursos disponíveis cada servidor possui e é atualizado por troca de mensagens entre os servidores.

Ao consultar o Mapa de Disponibilidade, é verificado se mais de um servidor pode atender a uma requisição. O componente fuzzy é acionado para decidir qual servidor irá receber este pedido. Esta decisão é baseada em duas variáveis: a latência entre o servidor que recebeu a requisição e cada servidor candidato a receber esta requisição, e a capacidade de processamento que cada servidor possui. Uma lista dos servidores com o maior número de recursos livres é criada, neste caso, e aplica-se a solução fuzzy nesta lista para a tomada de decisão para qual servidor deve ser encaminhada a requisição.

Se a latência entre dois servidores é a menor entre todos, isto não significa que seja a melhor opção, pois o servidor de destino pode estar com uma carga maior. Este algoritmo leva a um balanceamento de carga das solicitações de recursos, pois distribui estas requisições entre servidores que podem atender estas requisições, procurando responder a estas requisições eficientemente, em um menor tempo de resposta.

Algoritmos descentralizados, no aspecto de falhas, tem vantagem em relação àqueles centralizados, pois se um dos servidores falhar o sistema continua em funcionamento.

Raynal propôs um algoritmo descentralizado (Raynal, 1991), onde organiza o uso de *M* recursos, compartilhados por *N* processos, onde cada recurso pode ser utilizado por apenas um processo ao mesmo tempo, e cada processo pode solicitar novos recursos apenas após liberar os recursos em utilização.

Diferentemente da proposta de Raynal, o algoritmo deste trabalho discorre o uso de *M* recursos que estão distribuídos em *K* servidores, em que cada servidor possui m_i recursos (i de 1 a K) onde *N* usuários podem fazer requisições para cada servidor,

podendo ser feitas novas requisições antes mesmo do término de atendimento das solicitações anteriores.

Nas próximas seções segue-se uma descrição geral do algoritmo, os conceitos de Primeira Fila, Relógio Lógico e Mapa de Disponibilidade, do funcionamento do programa principal e suas rotinas de apoio.

4.1 Descrição geral do algoritmo

São considerados *k* servidores, cada um com *n* recursos. Cada servidor pode solicitar de *1* a *k*n* recursos. Não é necessário que, para efetuar uma nova solicitação, o servidor deva liberar os recursos que está utilizando.

Fazem parte deste algoritmo uma fila local de solicitação de recursos e o relógio lógico, que é utilizado para organizar as filas nos servidores. O relógio lógico é uma estrutura local em cada servidor, mas com um funcionamento global, dando a noção de um relógio único para todo o sistema, organizando as filas locais como uma única fila global, evitando *deadlock* e *starvation*.

A somatória de recursos individuais gerando M recursos globais tem como objetivo produzir um ambiente único para a realização de um trabalho.

Na Figura 4 tem-se a representação da distribuição destes recursos, da fila local de pedidos e do relógio lógico

Cada servidor, que tem recursos para compartilhar, recebe uma identificação (ID), que é o valor sequencial de entrada no grupo de servidores que irão compartilhar recursos. Este valor inicia em zero e é incrementado a cada servidor que entra no grupo.

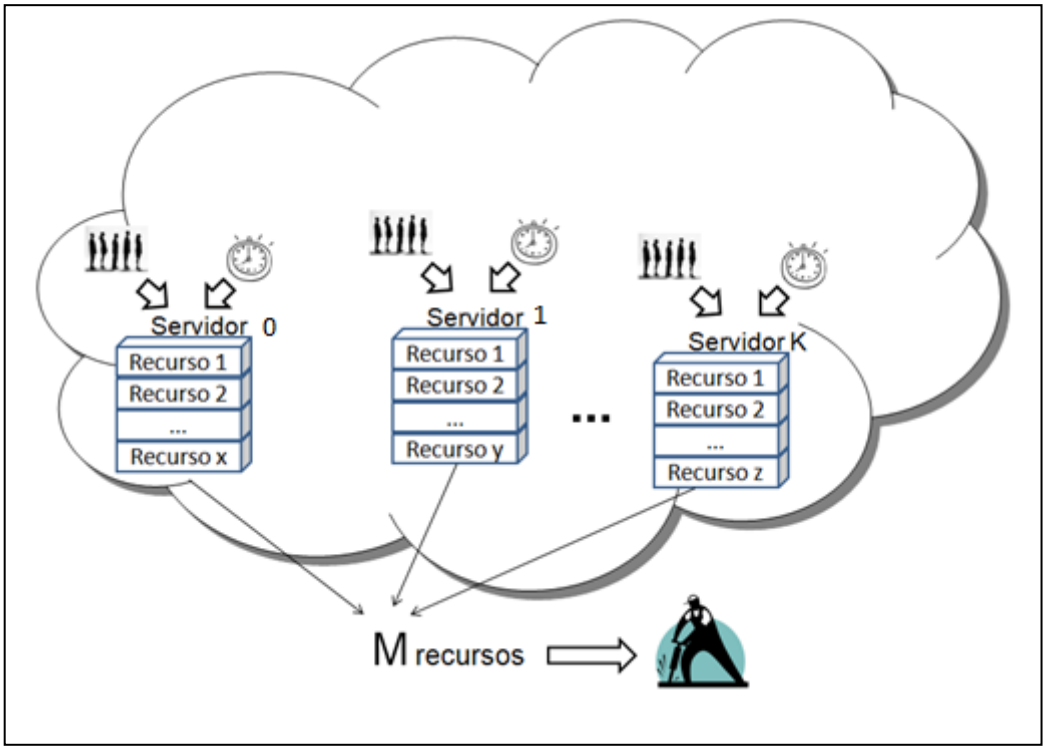

Figura 4 - Representação da distribuição de recursos (elaborado pelo autor, 2013)

A nuvem, nesta figura, representa a rede de interconexão que une os servidores de recursos participantes deste processo, que pode ser desde a ligação destes servidores em um *cluster*, até a união dos mesmos pela Internet.

Na próxima seção explica-se a fila local de pedidos, também chamada de REQ_Q.

4.1.1 Fila de Pedidos

Esta fila local é gerada em cada servidor, e armazena todas as solicitações de recursos enviadas para este servidor. Pedidos não atendidos ficam armazenados nesta fila, e são atendidos conforme os recursos que estavam sendo utilizados por outros processos sejam liberados.

Durante a execução do algoritmo, a dinâmica do funcionamento se dá com pedidos de recursos sendo feitos aos servidores, que pode gerar uma sobrecarga, dando origem a ocorrência de pedidos não atendidos. Em cada servidor participante do

compartilhamento há uma primeira fila de pedidos, que é um vetor de estrutura de dados, conforme descrito na Figura 5:

```
struct info_PF {
  int recs_pedidos;
  int quem_pediu;
  int lrelogio;
  int flag_atendido;     /* se o pedido foi processado este
                            campo fica = 1 */
};
struct info_PF REQ_Q[MAXPEDIDOS];
```

Figura 5 - Fila local de pedidos (elaborado pelo autor, 2013)

Esta estrutura de dados recebe os pedidos feitos ao servidor, que são enfileirados para serem processados. As variáveis desta estrutura tem o seguinte significado:

- recs_pedidos Contém a quantidade de recursos solicitados
- quem_pediu Qual servidor solicitou recs_pedidos
- lrelogio Valor do relógio lógico local no momento do recebimento da solicitação
- flag_atendido Se o pedido foi processado este campo fica igual a 1

Todo pedido recebido pelo servidor é armazenado nesta estrutura para o posterior tratamento desta solicitação. Quando um servidor recebe uma requisição, verifica se ele próprio possui recursos para atende-la. Se não possuir, esta é enviada a outro servidor. Caso não tenha recursos em nenhum servidor, a solicitação é colocada na fila do servidor que originalmente recebeu a solicitação.

Na próxima seção tem-se a explicação do funcionamento do relógio lógico.

4.1.2 Relógio Lógico

O relógio lógico é utilizado para garantir uma ordem global para todas as filas locais em cada servidor. Com a utilização deste relógio, é feito o controle para não ocorrer *deadlock* ou *starvation*, não permitindo que algum pedido fique eternamente na fila esperando recursos.

As variáveis *quem_pediu* e *lrelogio*,da Figura 5, compõe o relógio lógico, formando a tupla <*i, relogio*$_i$>, conforme descrito no item 2.1, sendo *quem_pediu* correspondendo ao valor *i*, e *lrelogio* ao valor *relogio*$_i$.

Quando há liberação de recursos, o relógio lógico é utilizado para decidir para qual solicitação o recurso deve ser alocado. O seguinte pseudo-código mostra o funcionamento do relógio lógico (Garg, 2002):

```
var c inteiro com valor inicial 0;
evento de envio()
        c=c+1;
        envia mensagem(mensagem, c)
evento de recebimento()
        recebe mensagem(mensagem(u))
        c=max(c,u.c)+1;
```

Ao enviar uma mensagem, o valor do relógio lógico é incrementado em 1 e incluído na mensagem. Ao receber uma mensagem, o valor do relógio lógico local é comparado com o valor recebido, e o maior valor entre eles é armazenado no relógio lógico local e incrementado em 1.

4.1.3 Funcionamento do relógio lógico

Nesta seção é feita uma discussão a respeito do funcionamento do relógio lógico. Nas figuras, o círculo maior representa um servidor, os círculos internos representam recursos. As setas representam mensagens. Ao lado de cada seta, segue a legenda da mensagem (1A, 1B, etc.) e o relógio lógico na forma (h_i, i) onde h_i é o valor local do relógio lógico e i é o número do cluster com este relógio lógico.

A situação inicial é representada por 4 servidores, cada um com 4 recursos, todos estes recursos estão disponíveis para utilização. Estes servidores estão ligados em rede, representando uma grade. Cada servidor pode utilizar seus próprios recursos e os recursos dos outros *clusters*, conforme a Figura 6.

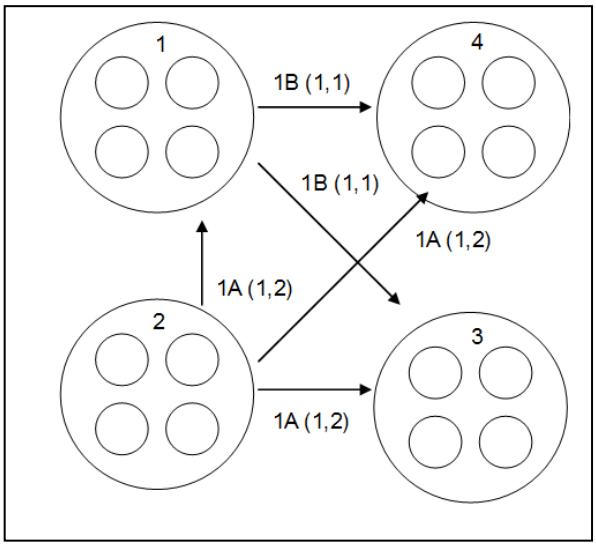

Figura 6 - Situação 1 (elaborado pelo autor, 2013)

Na Figura 6, tem-se a seguinte situação: 2 quer 4 recursos. Tem 4 recursos locais e com a mensagem 1A avisa a todos que quer usar os recursos próprios. 1 também quer 4 recursos. Tem 4 recursos locais e com a mensagem 1B avisa a todos que quer usar os recursos próprios. Primeiramente, 2 e 1 verificam suas filas locais; se estiverem vazias, então devem consultar seus vizinhos, pois eles podem ter pedidos pendentes (1 ou 2 podem ter entrado na grade após 3 e 4).

3 e 4 respondem a 1 e 2 que não tem pedidos pendentes em suas filas locais (mensagens 2A e 2B); o mesmo fazem 1 e 2 entre si (mensagens 2C e 2D), conforme a Figura 7.

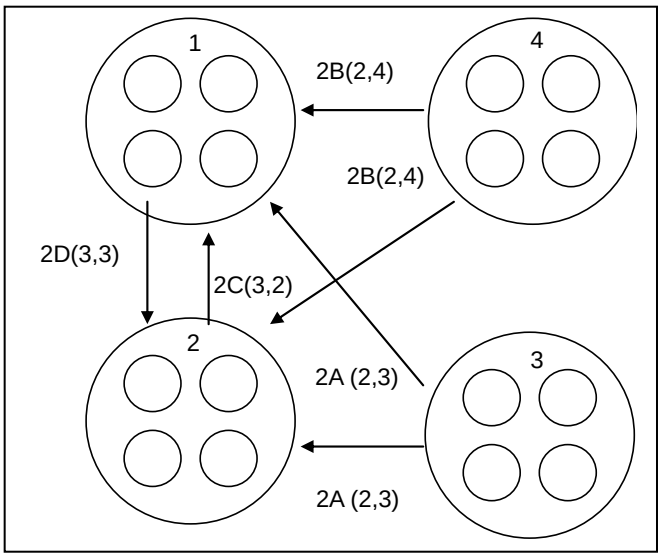

Figura 7 - Situação 2 (elaborado pelo autor, 2013)

2 e 1 precisam de mais 8 recursos. 2 envia a mensagem 3A e 1 a mensagem 3B solicitando estes recursos. Esta situação está ilustrada na Figura 8.

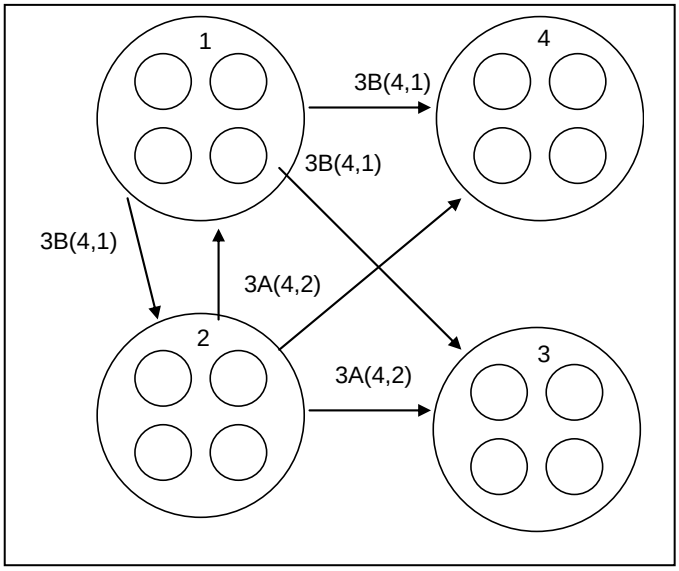

Figura 8 - Situação 3 (elaborado pelo autor, 2013)

Os servidores 3 e 4 recebem as duas mensagens solicitando seus recursos. O servidor 3 decide pelo servidor 1, devido ao relógio lógico. O mesmo acontece com o servidor 4. Uma mensagem é enviada pelo servidor 3 e pelo servidor 4 informando esta decisão, como mostrado na Figura 9.

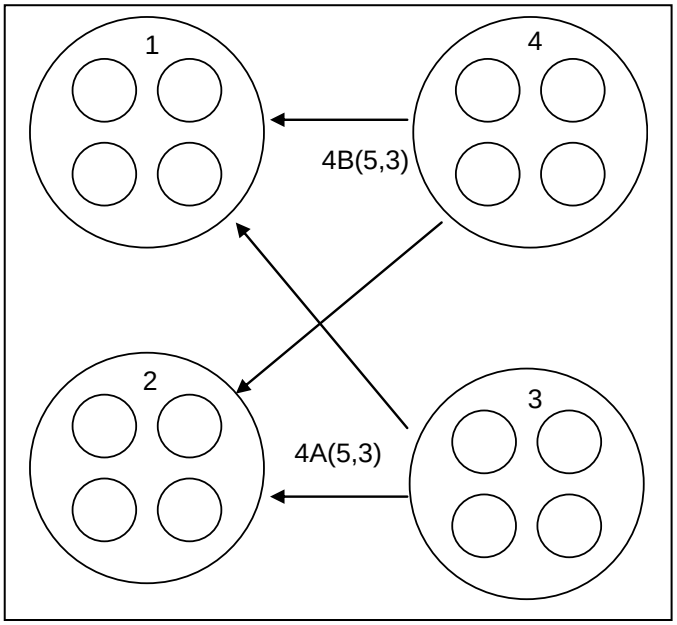

Figura 9 - Situação 4 (elaborado pelo autor, 2013)

A solicitação do servidor 2 fica aguardando a liberação de recursos para ser atendida.

Na próxima seção tem-se a explicação do Mapa de Disponibilidade.

4.2 Mapa de Disponibilidade

O Mapa de Disponibilidade é uma estrutura de dados utilizada no mecanismo de verificação de qual servidor é elegível para receber requisições e é mantido atualizado por troca de mensagens. Esta estrutura de dados está representada na Figura 10.

Cada posição do vetor contém informações de um servidor participante do processo de compartilhamento de recursos. As variáveis desta estrutura são:

- MD Representa a quantidade de recursos livres que o servidor contém
- relógio Um contador inteiro com o valor do relógio lógico deste servidor
- flag Se flag=1, este servidor tem pedidos de recursos pendentes a serem atendidos; se flag=0 a sua fila local de pedidos está vazia.

```
struct info_MD {
     int MD;
     int relogio;
     int flag;
};
struct info_MD MD[N_NOS];
```

Figura 10 - Mapa de Disponibilidade (elaborado pelo autor, 2013)

Quando um servidor precisa de recursos, ele verifica se há recursos locais que possa atender a solicitação e se não há pedidos pendentes em outros servidores. Pelo relógio lógico é feito o desempate, caso ocorra o fato de outro servidor tiver pedidos pendentes. isto ocorre quando a variável *flag*, da Figura 10, está com valor igual a 1.

Na próxima seção detalha-se a estrutura do algoritmo.

4.3 Estrutura do algoritmo

Cada servidor executa quatro *threads*: trata_MSG, aloca_RECS, recs_PEDIDOS, e ack_PENDENTE. Um cliente envia uma requisição executando a função CLIENTE. Na fase de inicialização, os servidores trocam mensagens entre si para sincronização de informações e atualização do Mapa de Disponibilidade.

Ao iniciar sua execução, o programa principal ativa as três primeiras *threads*. A primeira rotina, trata_MSG é responsável pelo recebimento de mensagens que tem como objetivo a atualização de dados entre os servidores e o recebimento de solicitações. A segunda rotina, Aloca_RECS, trata da solicitação de recursos e ativa os módulos para decisão de quem vai atender a solicitação. A terceira rotina, recs_PEDIDOS, é consultada quando um servidor quer permissão para utilizar recursos e devolve um reconhecimento positivo ou negativo a quem a consultou.

Outras duas rotinas são ativadas pelas *threads* somente quando necessário. A primeira é a rotina Fuzzy, utilizada para auxiliar na decisão quando há mais de um candidato a atender uma solicitação, e a segunda é rotina processa_RECS, ativada para processar a solicitação de recursos.

O algoritmo usa onze tipos de mensagens com os seguintes parâmetros:

c identifica o cliente que envia a requisição de recursos.

s identifica o servidor onde o cliente está conectado.

r identifica o servidor remoto que possui recursos livres.

l valor do relógio lógico do servidor no momento que o mesmo recebe a requisição de recursos.

a reconhecimento positivo ou negativo com relação a disponibilidade de recursos.

p o número de requisições pendentes locais a um servidor com relógio lógico menor do que o valor do relógio de uma requisição l de recursos recebida.

nrecs quantidade de recursos solicitados pelo cliente

As mensagens tem a seguinte estrutura:

<REQUISITA, c >: o cliente c envia uma requisição de recursos para o servidor s com quem está conectado;

<LIBERA, r >: enviado pelo servidor r para todos servidores ao término da execução da requisição com o objetivo de liberar este recursos;

<SOLICITA_RECS, nrecs, r> enviado ao servidor r, candidato a oferecer nrecs recursos para quem enviou esta mensagem;

<RECURSOS, s >: enviado pelo servidor s depois de ter alocado recursos no servidor r para uma requisição de um cliente;

<PENDENTE, r, l, s >: enviado pelo servidor s para todos os servidores quando é necessário alocar recursos no servidor r. Esta mensagem solicita uma confirmação de que há recursos para satisfazer a requisição do servidor s e todas as requisições pendentes com prioridade maior do que a requisição enviada ao servidor s;

<RESPOSTA_ACK, a, p >: reconhecimento positivo ou negativo enviado por um servidor x como uma resposta a uma mensagem <PENDING> recebida do servidor s. O servidor x, nesta resposta, também inclui o valor p, que representa o total de requisições pendentes com prioridade maior do que a requisição enviada pelo servidor s;

<FINALIZANDO, c >: enviada pelo servidor r para o servidor s com o resultado da requisição do cliente c;

<RESULTADO, resultado > mensagem enviada pelo servidor s para o cliente c com o resultado do processamento da requisição;

<OK_PROCESSA,i,s>: enviada ao servidor s o valor i. Se i igual a 0 indica que a solicitação de recursos foi recusada. Se igual a 1 indica que o servidor utilizará os recursos solicitados;

<ATUALIZA_MD_OK, cont>: enviada a todos os servidores para atualização do Mapa de Disponibilidade;

<ACK_LOCATED, 1, quem_pediu_recs>: enviada pela tread recs_PEDIDOS para informar todos os servidores que um determinado servidor vai utilizar seus recursos.

A rotina trata_MSG() é a rotina responsável pelo recebimento das mensagens enviadas pelos outros servidores. Ao enviar uma mensagem, as variáveis são

concatenadas em uma sequência única de caracteres, formando um pacote de dados, composta pelas seguintes variáveis inteiras, conforme a Tabela 3:

Tabela 3 – Variáveis de uma mensagem

quem_mandou	Contém a identificação do servidor que enviou a mensagem
recs_disponíveis	É a quantidade de recursos disponíveis no servidor que enviou a mensagem
relógio	Valor do relógio lógico de quem enviou a mensagem
liberando_recs	Se igual a -3, a mensagem está solicitando recursos Se igual a -1, informa que foi alocado recursos Se igual a 1, está liberando recursos Se igual a 3, informa que há pedidos na fila local
recs_pedidos	Informa quantos recursos estão sendo solicitados

Estas variáveis são globais para cada servidor. No recebimento de uma mensagem elas são atualizadas com os valores recebidos. A variável liberando_recs indica ao servidor que recebeu a mensagem qual é o objetivo da mensagem.

Três filas são mantidas por cada servidor: REQ_Q, PEND_Q, e RES_Q. Elas respectivamente armazenam as mensagens das *threads* CLIENTE, . Se não há mensagem na fila, a *thread* fica bloqueada; senão a *thread* passa para o estado de execução.

A *thread* Trata_MSG recebe mensagens de clientes locais e dos servidores remotos, redirecionando para as *threads* correspondentes, colocando cada mensagem em sua fila correspondente ou enviando as mesmas para os clientes.

Com o objetivo de evitar starvation, toda requisição recebida por um servidor s é assinalada com o valor corrente do relógio lógico do servidor s. Antes de incluir a mensagem na fila RES_Q, a *thread* Trata_MSG incrementa seu relógio lógico. Uma ordem total das requisições pode ser estabelecida baseada no valor dos relógios lógicos assinalados e, se necessário, a identificação do servidor é utilizada em caso de empate no valor do relógio lógico.

Em outras palavras, uma requisição cujo valor do relógio lógico associada a ela é menor do que uma segunda requisição tem prioridade sobre ela. Se os valores dos relógios lógicos são iguais, a requisição do servidor com identificador menor tem a prioridade no atendimento. Em ambos os casos, diz-se que a mensagem com a requisição com maior prioridade tem precedência sobre a outra. Por isso, as mensagens <REQUISITA> na fila RES_Q são ordenadas pela prioridade para

assegurar que toda mensagem será atendida e portanto satisfeita. Por uma questão de simplicidade, a associação do relógio lógico a uma mensagem não foi incluída na notação das mensagens.

Nas solicitações de recursos o pedido é incluído na Fila RES_Q, que é um vetor com a estrutura representada pela Tabela 4.

Tabela 4 – Estrutura das Filas Locais

recs_pedidos	Contém a quantidade de recursos solicitados
quem_pediu	Indica qual servidor fez a solicitação
lrelogio	Valor do relógio lógico de quem enviou a mensagem
liberando_recs	Se igual a -3, a mensagem está solicitando recursos Se igual a -1, informa que foi alocado recursos Se igual a 1, está liberando recursos Se igual a 3, informa que há pedidos na fila local
flag_atendido	Se igual a 0, esta solicitação ainda não foi atendida

A rotina aloca_RECS é o mecanismo responsável pela alocação de recursos que foram solicitados ao servidor que está executando esta cópia do algoritmo. Após verificar, pelo relógio lógico, se este pedido é prioritário para esta solicitação, é feita a verificação se este servidor tem recursos próprios para atender a solicitação. Caso não tenha, é verificado onde há recursos livres. Se apenas um servidor tem recursos, então utiliza-se seus recursos. Senão, a rotina fuzzy é chamada passando-se dois valores: a latência de rede entre o servidor que recebeu a solicitação e o valor da potência que o servidor candidato tem no momento da chamada da rotina.

O valor retornado pela rotina fuzzy é armazenado em um vetor. O maior valor deste vetor indica o servidor que deve atender a solicitação.

A *thread* Aloca_RECS manipula as mensagens <REQUISITA,c>. Para cada uma destas mensagens, é verificado se o servidor local tem recursos disponíveis. Se não é o caso, é verificado se apenas um servidor pode atender a solicitação. Se mais de um servidor pode atender, a decisão para qual servidor deve ser encaminhada a requisição é feita pela lógica fuzzy.

Entretanto, antes de utilizar o recurso do servidor indicado pela função fuzzy, é necessário assegurar que o recurso pode ser alocado pela requisição em questão. Tal procedimento é necessário, pois, devido aos pedidos simultâneos em outros servidores, o recurso indicado pela rotina fuzzy pode ser alocado para executar um

segundo pedido no mesmo período. Portanto, uma mensagem <PENDENTE, r, l, s> é enviada a todos os servidores para confirmar que o recurso pode ser utilizado.

Ao receber uma mensagem <PENDENTE, r, l, s>, a *thread* Ack_PENDENTE do servidor r calcula o número de recursos solicitados em requisições locais pendentes cujo relógio lógico tem prioridade sobre a requisição enviada pelo servidor s com relógio lógico l. Estas informações são enviadas ao servidor s em uma mensagem <RESPOSTA_ACK, a, p>. O servidor s pode utilizar estes recursos se há recursos suficientes no servidor r tanto para o seu próprio pedido e de todos os outros pedidos que tem prioridade sobre o seu pedido. Caso contrário, o servidor s deve alocar outro recurso e solicitar novamente a permissão dos outros servidores.

A *thread* Processa_RECS é responsável por executar a solicitação. Após sua utilização, o recurso é liberado e o resultado enviado ao cliente.

A Figura 11 ilustra o funcionamento do algoritmo quando o Servidor 1 recebe uma solicitação de recursos mas não tem recursos próprios, nenhum servidor tem recursos suficientes para atender integralmente, e vários servidores devem contribuir com recursos para efetivar o atendimento desta requisição.

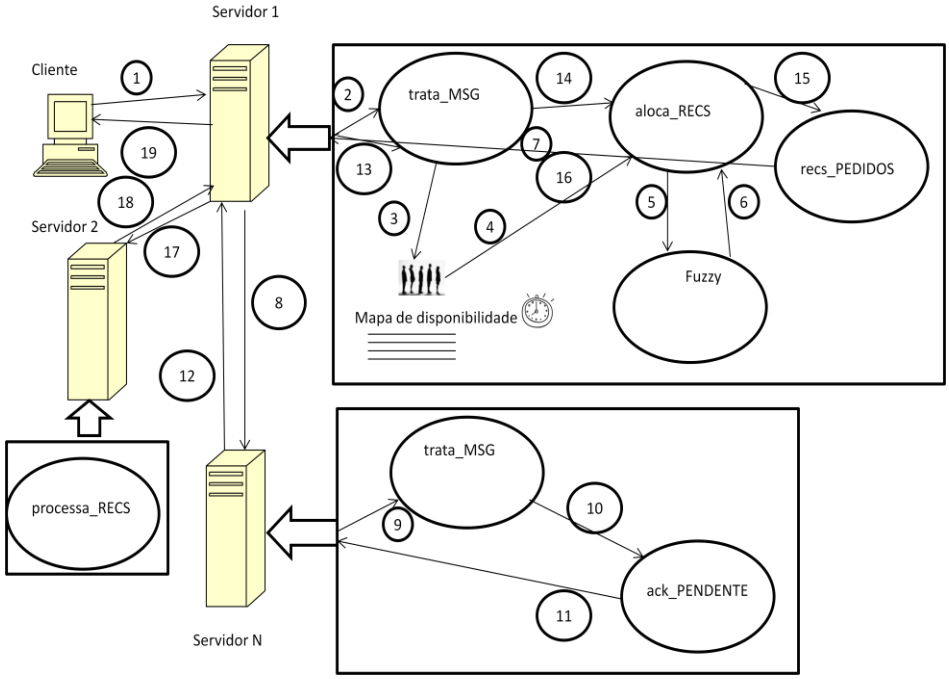

Figura 11 - Solicitação de recursos (elaborado pelo autor, 2013)

Uma explicação de cada passo desta situação é mostrada a seguir.

1 – Cliente envia uma solicitação de recursos

2 – A requisição é recebida pela *thread* trata_MSG

3 – A requisição é armazenada na fila REQ_Q

4 – A *thread* aloca_RECS verifica a fila REQ_Q e trata a solicitação não atendida

5 – Verificando o Mapa de Disponibilidade, a *thread* aloca_RECS verifica que nenhum servidor tem recursos para atender a solicitação.

6 – A rotina Fuzzy é acionada para cada servidor candidato, onde a latência entre o Servidor 1 e o servidor candidato e o valor da potência que o candidato tem são passados para a rotina Fuzzy.

7 – Como mais de um servidor vai contribuir com recursos para atender a solicitação, uma lista destes candidatos é gerada, com o valor de recursos que cada um vai disponibilizar para o completo atendimento da requisição.

8 – Antes de requisitar recursos aos servidores escolhidos, uma mensagem é enviada a todos os servidores solicitando autorização para a utilização dos recursos dos candidatos.

9 – Cada servidor recebe a mensagem pela *thread* trata_MSG que a armazena na fila PEND_Q.

10 – A *thread* ack_PENDENTE verifica a solicitação na fila PEND_Q e se não há solicitações pendentes no servidor que executa esta *thread*

11 – Uma resposta positiva, indicando que os recursos podem ser utilizados por quem solicitou, ou negativa é enviada em resposta a mensagem ack_PENDENTE

12 – O servidor 1 aguarda a resposta ack_PENDENTE de todos os servidores

13 – As respostas são recebidas pela *thread* trata_MSG.

14 - Caso todas as respostas sejam positivas, a mensagem é incluída na fila RES_Q.

15 – A *thread* recs_PEDIDOS verifica a fila RES_Q

16 – O Servidor 1, tendo autorização de todos os servidores, prepara uma mensagem para os Servidores escolhidos para atender a solicitação

17 – A solicitação é enviada aos Servidores, que ativam a *thread* processa_RECS, executando a solicitação de recursos.

18 – A resposta do processamento é enviada ao Servidor 1.

19 – O cliente recebe o resultado do processamento.

4.4 Estratégia adotada para o funcionamento do algoritmo de busca de recursos

Esta pesquisa levou ao desenvolvimento de uma estratégia para alocação de recursos distribuídos, conectados por uma rede de computadores. Um grupo de servidores, cada um com um determinado número de recursos, trocam mensagens via socket, utilizando o modelo TCP/IP. Estas informações trocadas alimentam o algoritmo implementado, que tem sua funcionalidade distribuída em *threads*. Esta modularidade permite uma flexibilização na execução do algoritmo, com o objetivo de otimizar a alocação destes recursos.

Esta estratégia leva em conta o seguinte aspecto antes de um recurso ser utilizado com o objetivo de satisfazer uma requisição: o servidor que recebeu a requisição deve consultar os demais servidores que compartilham recursos sobre a possibilidade da utilização dos mesmos. Um processo de votação é iniciado pelo envio de uma mensagem requisitando esta votação. O recurso só é utilizado após o recebimento de uma resposta positiva sobre este questionamento.

Após receber uma resposta positiva de todos os outros participantes, uma somatória de todas as requisições com relógio lógico com maior prioridade do que a requisição em questão garante que nenhuma requisição anterior fique esperando eternamente pelo atendimento, caso ocorra um atraso na transmissão de uma mensagem pela rede.

A estratégia proposta permite a alocação de apenas um servidor, no caso em que dois ou mais servidores selecionam o mesmo servidor após obterem simultaneamente a disponibilidade de recursos suficientes em seus respectivos Mapas de Disponibilidades, sendo, entretanto, insuficientes para todos.

Para evitar este possível conflito, foi desenvolvida a estratégia descrita a seguir, e o algoritmo está localizado no item 4.5.

4.4.1 Seleção dos recursos

A rotina aloca_RECS é responsável pela verificação, na fila de pedidos, de requisições não atendidas, na procura de recursos livres para atender estas solicitações e o encaminhamento destas requisições a servidores remotos, isto se não houver recursos locais para efetivar o atendimento.

Esta rotina é formada por quatro partes, todas com o mesmo objetivo, mas com funcionalidades diferentes. A primeira parte procura por recursos no próprio servidor que recebeu a solicitação de recursos por um cliente. A segunda parte verifica, caso a primeira parte não satisfaça a requisição, se apenas um servidor pode atender a mesma. A terceira é acionada caso as duas partes anteriores não atendam a solicitação, e verifica se mais de um servidor pode atender a solicitação. Caso isto ocorra, nesta parte da rotina é acionado o módulo fuzzy para auxiliar na decisão para qual servidor com recursos será encaminhada a solicitação.

A última parte da rotina aloca_RECS procura atender a solicitação de recursos caso nenhum servidor possa atender integralmente a solicitação, mas há recursos suficientes distribuídos para executar o pedido do cliente.

4.4.2 Autorização para uso dos recursos

Se tudo está correto e não ocorreu erro, inicia-se o processo de votação, onde é solicitada autorização de todos para a utilização destes recursos com o envio de uma mensagem para todos os servidores.

A variável contALOCAsim é incrementada a cada resposta ACK positiva e os recursos poderão ser utilizados caso todos os servidores enviem esta resposta ao servidor que solicitou a votação.

Neste processo de votação, onde a *thread* ack_PENDENTE é acionada, o servidor que recebe esta mensagem verifica em sua fila local se há pedidos pendentes com relógio lógico menor do que o relógio do pedido em análise. A variável *soma*, na *thread* ack_PENDENTES armazena este resultado, que é devolvido ao servidor que solicitou a votação, é acumulada na variável *number_resources_pending*, que no

final da votação conterá a soma de todas requisições com relógio lógico menor ao pedido em questão.

Se a soma local de recursos solicitados com relógio lógico menor ao pedido enviado, uma resposta negativa a solicitação é enviada pela *thread* ack_PENDENTE, impossibilitando a utilização dos recursos solicitados. O objetivo desta negação é de impedir uma espera eterna, ou starvation, no servidor que recebeu a mensagem pois poderia ocorrer uma situação onde um pedido ainda não avaliado localmente não seja atendido e os recursos sejam utilizados por outro pedido com relógio lógico com menor prioridade, pedido este de outro servidor.

Após receber a resposta ack_PENDENTE de todos os servidores, o processo de alocação continua caso todas as respostas recebidas seja ACK_YES, que significa que não há impedimento local, por quem enviou esta resposta, na utilização dos recursos solicitados.

A variável *number_resources_pending* indica se os recursos poderão ou não ser utilizados. Caso este valor seja maior do que o total de recursos disponíveis em todos os servidores mais o número de recursos solicitados, o processo de alocação é abortado, também para se evitar que algum pedido local em outro servidor, com relógio lógico com maior prioridade fique esperando eternamente por recursos.

Para cada servidor escolhido para fornecer recursos para o atendimento total da requisição, é enviada uma mensagem <SOLICITA_RECS, nrecs, r>. Com o objetivo de se evitar a situação onde dois servidores enviam a mensagem SOLICITA_RECS para o servidor r ao mesmo tempo, como resposta a esta solicitação, o servidor r verifica se ele possui recursos suficientes para atender esta solicitação e envia, para cada requisição recebida, uma resposta positiva ou negativa a esta consulta.

Isto é feito para se evitar que dois servidores, ao consultar o Mapa de Disponibilidade ao mesmo tempo, tenham a visão de que há recursos para atender a sua solicitação, e que duas solicitações de recursos sejam enviadas simultaneamente a um mesmo servidor. Estas solicitações são armazenadas na fila RES_Q de quem recebeu a requisição.

Portanto, o servidor que envia a mensagem SOLICITA_RECS aguarda uma confirmação de cada servidor candidato a atender parcialmente a solicitação de recursos. Se todas as respostas forem positivas, os recursos podem ser utilizados e

isto garante que apenas uma requisição será atendida, evitando que duas requisições que sejam enviadas simultaneamente por diferentes servidores possam causar problemas na alocação de recursos indisponíveis.

Caso o próprio servidor que recebeu a requisição de recursos por um cliente tenha recursos disponíveis, um tratamento especial é feito para que o sistema utilize seus recursos como parte da solução de alocação dos recursos solicitados. Nenhuma mensagem é enviada, pois não há necessidade de comunicação com outros servidores para se utilizar recursos próprios.

4.4.3 Atualização do Mapa de Disponibilidade

A atualização do Mapa de Disponibilidade ocorre em dois momentos. Primeiro quando os recursos de um servidor são utilizados e a quantidade de recursos é decrementada no Mapa de Disponibilidade. Segundo quando os recursos são liberados após a sua utilização e esta quantidade de recursos é incrementada no Mapa de Disponibilidade.

Feita a escolha dos servidores que vão atender a solicitação de recursos, uma mensagem <SOLICITA_RECS, nrecs, r> é enviada a cada um dos candidatos a ceder recursos. O servidor que recebe esta mensagem, verifica se tem recursos para atender a solicitação. Se tiver, envia ao solicitante de recursos a mensagem <ACK_LOCATED, 1, quem_pediu_recs>.

O servidor solicitante espera pela resposta <ACK_LOCATED> enviada pelo servidor que vai ceder recursos. Sendo esta mensagem com valor 1, o solicitante envia a mensagem <ATUALIZA_MD_OK, nrecs> para todos os servidores atualizarem seus Mapas de Disponibilidade decrementando o valor *nrecs*, que corresponde ao valor de recursos utilizados.

Uma mensagem <OK_PROCESSA, 1> é enviada para cada servidor que vai disponibilizar recursos para atender a solicitação. Esta mensagem ativa a *thread* recs_PEDIDOS de cada um destes servidores, iniciando o processo de utilização dos recursos disponibilizados. No caso da utilização de recursos próprios, a ativação é direta no servidor que está solicitando recursos, com a criação da *thread* local processa_RECS.

Caso algum servidor não confirme a disponibilidade de recursos, uma mensagem <OK_PROCESSA, 0> é enviada e o processo de alocação é abortado.

Após a utilização dos recursos, a *thread* processa_RECS é responsável pela atualização do Mapa de Disponibilidade. Uma mensagem <ATUALIZA_MD_OK> é enviada a todos os servidores antes da finalização desta *thread*, informando a todos os servidores que os recursos que estavam ocupados agora estão disponíveis para utilização, e outras solicitações pendentes podem ser atendidas.

4.4.4 Recebendo mensagens

Toda mensagem recebida é tratada pela *thread* trata_MSG, que é responsável por dividir a mensagem passando as devidas variáveis para as respectivas *threads*, incluindo nas filas locais REQ_Q as solicitações enviadas pelo cliente, PEND_Q as mensagens enviadas para a *thread* ack_PENDENTE e RES_Q as requisições que são recebidas direcionadas para a *thread* recs_PEDIDOS. A atualização do Mapa de Disponibilidade é feita nesta *thread*, incluindo a contabilização das respostas positivas ou negativas enviadas pela *thread* ack_PENDENTE.

4.4.5 Executando solicitação

A *thread* recs_PEDIDOS auxilia a *thread* aloca_RECS recebendo as solicitações de requisições, verificando a fila RES_Q, enviando para a *thread* processa_RECS as solicitações autorizadas a serem executadas. Antes de executar a solicitação, uma mensagem <ACK_LOCATED, 1,quem_pediu_recs> é enviada para garantir que apenas um servidor vai utilizar seus recursos.

A Figura 12 ilustra a seguinte situação: O servidor 1 recebeu uma solicitação de recursos e precisa, para atender esta solicitação, recursos do servidor 2, do servidor 3 e do servidor 4.

A troca de mensagens entre os servidores tem a seguinte descrição:

1) O servidor 1 envia a mensagem <SOLICITA_RECS> para os servidores 2, 3 e 4. Estes servidores armazenam esta solicitação em suas filas locais RES_Q

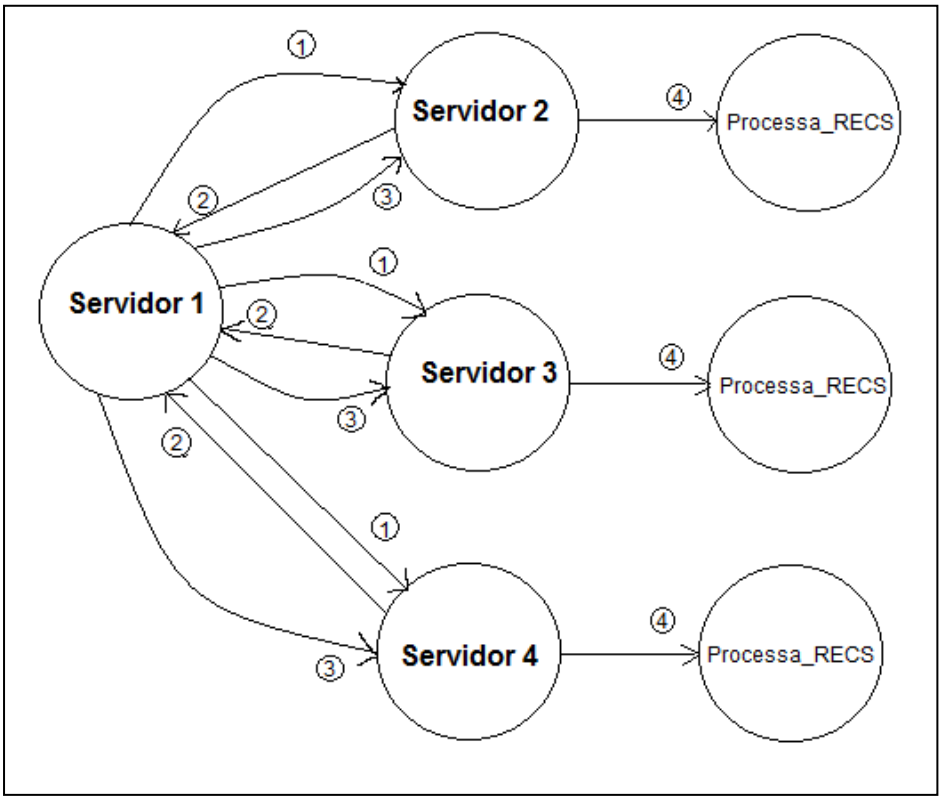

Figura 12 - Garantindo recursos (elaborado pelo autor, 2013)

2) Cada servidor que recebeu a mensagem <SOLICITA_RECS> verifica sua disponibilidade de recursos. Caso seja possível atender a solicitação, é enviada uma mensagem <ACK_LOCATED, 1> para o servidor 1
3) Ao receber a mensagem <ACK_LOCATED, 1> dos três servidores, o nó 1 atualiza o Mapa de Disponibilidade e envia uma mensagem <OK_PROCESSA, 1> para os servidores 2, 3 e 4
4) Ao receberem a mensagem <OK_PROCESSA, 1>, os servidores 2, 3 e 4 ativam a *thread* processa_RECS, iniciando o processamento nos recursos solicitados

4.5 Algoritmo para alocação de recursos

```
1
2     /* Função chamada pelos clientes */
3     CLIENTE (c)
4     { /* esta rotina é utilizada por um cliente para
5          solicitar um recurso a um servidor */
6        envia <REQUISITA, c> para um servidor local;
7        recebe <msg> /* msg RESULTADO */
8     }
```

```
 9
10    /* Algoritmo executado pelos servidores */
11
12    principal ( )
13    {
14       Criar thread Trata_MSG;
15       Criar thread aloca_RECS;
16       Criar thread recs_PEDIDOS;
17       Criar thread ack_PENDENTE;
18    }
19
20   Trata_MSG(sid)
21    { /* esta thread espera por mensagens de
22      outros servidores e de clientes */
23      enquanto (verdade) {
24        recebe(<msg>);
25        troca (<msg.t>){
26        REQUISIÇÃO:
27           l++;
28          insere_fila (REQ_Q, <REQUISIÇÃO,
29            msg.c,msg.l>);
30        PENDENTE:
31            insere_fila (PEND_Q,<msg>);
32       RESPOSTA_ACK:
33            se(resposta == ACK_SIM)
34                 contACKsim++;
35            se(resposta == ACK_NAO)
36                 contACKnao++;
37       RECURSOS:
38            insert_queue (RES_Q,<msg>);
39        FINALIZANDO:
40           Envia <RESULT,msg.res> para c;
41        LIBERANDO: atualiza_Mapa_Disponibilidade ();
42
43        }
44      }
45     }
46    int aloca_RECS()
47    enquanto (há pedidos a serem atendidos)
48    {
49       para(y=0;y<contREQ;y++)
50       {
51            nrecs=REQ_Q[y].recs_pedidos;
52            se(nrecs <= MD[id].MD)
53                 x1=id;
54                 recs_PEDIDOS(x1,nrecs);
55            senão
56            {
57                 cand=verifica_numero_candidatos();
58                 se(cand == 1 )
59                 {
60                      x1=candidato único;
61                      recs_PEDIDOS(x1,nrecs);
62                 }
```

```
63                    senão
64                    se(cand > 1)
65                    {
66                         sort_RF();
67                         cont=0;
68                         enquanto(cont<N_NOS)
69                         {
70                              se(MD[resultadoFUZZY[cont].MD>=nrecs)
71                              {
72                                   x1=resultadoFUZZY[cont].servidor;
73                                   recs_PEDIDOS(x1,nrecs);
74                                   cont=N_NOS;
75                              }
76                              cont++;
77                         }
78                    }
79                    senão
80                    para(z=0;z<N_NOS;z++) totrecs=totrecs+MD[z].MD;
81                    se(totrecs < nrecs)
82                    {
83                         erro=1;
84                         break;
85                    }
86                    se(cand == 0 e totrecs >= nrecs)
87                    {
88                         ultimo=N_NOS-1;
89                         enquanto (nrecs > 0)
90                         {
91                              sort_MD();
92                              x1=resultado[último].server;
93                              se(MD[x1].MD >= nrecs)
94                              {
95                                   usados=nrecs;
96                              }
97                              senão
98                              {
99                                   usados=MD[x1].MD;
100                             }
101                             PEND_LOCAL.raloc[x1]=usados;
102                             MD_sort[x1].MD=MD_sort[x1].MD-usados;
103                        }
104                        se(erro = 1)
105                        {
106                             y--;
107                             continua;
108                        }
109                        contALOCAsim=0;
110                        contALOCAnao=0;
111                        number_resources_pending=0;
112                        para(cont=0;cont<N_NOS;cont++)
113                        {
114                         envia <PENDENTE,r,l,s> para servidor cont;
115                        }
116                        enquanto(contALOCAsim+contALOCAnao<N_NOS-1)
```

```
117                         espera;
118                    se(contALOCAsim==N_NOS-1)
119                    {
120                         se(number_resources_pending >
121                              totrecs-nrecs_solicitado)
122                         {
123                              y--;
124                              PEND_LOCAL.flag_atendido=1;
125                              continua;
126                         }
127                    }
128                    senão
129                    {
130                         y--;
131                         PEND_LOCAL.flag_atendido=1;
132                         continua;
133                    }
134                    para(cont=0;cont<N_NOS;cont++)
135                    {
136                         LOCATED_YES[cont]=1;
137                         INFO_LOCATION[cont]=0;
138                    }
139                    para(cont=0;cont<N_NOS;cont++)
140                    se((PEND_LOCAL.raloc[cont] != 0) e
141                         (cont != id)
142                    {
143                         envia <SOLICITA_RECS,cont>
144                    }
145                    para(cont=0;cont<N_NOS;cont++)
146                    {
147                         se(cont != id) e
148                          (PEND_LOCAL.raloc[cont] != 0))
149                         {
150                          enquanto(INFO_LOCATION[cont]==0)
151                              espera;
152                         }
153                         se(LOCATED_YES[cont] ==0)
154                         {
155                              erro=1;
156                              break;
157                         }
158                    }
159                    se(erro=1)
160                    {
161                         para(cont=0;cont<N_NOS;cont++)
162                         {
163                              envia <OK_PROCESSA,0,cont>
164                         }
165                    }
166                    para(cont=0;cont<N_NOS;cont++)
167                    {
168                         envia <ATUALIZA_MD_OK,cont>
169                    }
170                    enquanto(ATUAL_DONE=0)
```

```
171                         espera;
172                    para(cont=0;cont<N_NOS;cont++)
173                    {
174                         envia <OK_PROCESSA, 1, cont>
175                    }
176                    para (cont=0;cont<N_NOS;cont++)
177                    {
178                         se(candidatos[id].aloca[cont]!=0)
179                         {
180                    recs=candidatos[id].aloca[cont].recursos;
181                         se(i != id)
182                         {
183                              x1=cont;
184                         }
185                         senão
186                         {
187                              x1=id;
188                         }
189                         recs_PEDIDOS(x1,usados);
190                    }
191               }
192          }
193       }
194    }
195  }
196  recs_PEDIDOS(x1,nrecs)
197  {
198    enquanto (há pedidos a serem atendidos)
199    {
200     para(y=0;y<contRESQ;y++)
201     {
202          se(RES_Q[y].flag_atendido=0)
203          {
204               nrecs=RES_Q[y].recs_pedidos;
205               quem_pediu_recs=RES_Q[y].quem_pediu;
206               para(cont=0;cont<N_NOS;cont++)
207               {
208                    envia <ACK_LOCATED, 1,quem_pediu_recs>;
209               }
210               enquanto(INFO_PROCESSAR=0)
211                    esperar;
212               se(OK_PROCESSAR = 1)
213               {
214                    cria thread processa_RECS;
215               }
216          }
217     }
218  }
219  void ack_PENDENTE()
220  {
221     para(x=0;x<=N_NOS;x++)
222     {
223          se(PEND_Q[x].flag_atendido == 0)
224          {
```

```
225                 para(y=0;y<contREQ;y++)
226                 {
227                      se ((REQ_Q[x].lrelogio<relogiopedido)
228                      ou (REQ_Q[y].lrelogio == relogiopedido)
229                      e  (id<x))
230                           RL=1;
231
232                      se ((REQ_Q[y].flag_atendido ==0
233                      e  (REQ_Q[y].recs_pedidos>0)
234                      e  (RL == 1)
235                           soma=soma+REQ_Q[y].recs_pedidos;
236                 }
237                 totrecs=0;
238                 flag=0;
239                 para (cont=0;cont<N_NOS;cont++)
240                 {
241                      totrecs=totrecs+MD[cont].MD
242                 }
243                 se(totrecs < soma + nrecs)
244                 {
245                      envia <ACK_YES,soma> para servidor x
246                 }
247                 senão
248                 {
249                      envia <ACK_NO, soma> para servidor x
250                 }
251            }
252       }
253  }
254  int processa_RECS(nrecs,pedido,quem_pediu_recs)
255  {
256       executa o processamento;
257       para(cont=0;cont<N_NOS;cont++)
258       {
259            envia <ATUALIZA_MD_OK,cont>
260       }
261       enquanto(ATUAL_DONE_PROCESSA=0)
262            espera;
263  }
264
```

4.6 Rotina FUZZY

A rotina fuzzy é acionada pela *thread* Aloca_RECS. O retorno desta rotina é um valor defuzificado que representa o resultado da utilização da teoria fuzzy.

Neste trabalho foram modelados dois conjuntos fuzzy de entrada e um conjunto fuzzy de saída. Os conjuntos de entrada correspondem a latência entre os servidores e a potência de processamento de cada servidor. A variável de saída

contém a modelagem correspondente aos valores de saída. Estes conjuntos estão apresentados na Tabela 5.

Tabela 5 - Conjuntos Fuzzy de entrada e saída

Latência			
Subconjunto	*Início*	*Fim*	*Abreviação*
Excelente	0.0	40.0	EX
Muito Bom	30.0	50.0	MB
Bom	40.0	100.0	BO
Regular	80.0	150.0	RE
Ruim	140.0	300.0	RU
Potência			
Excelente	1.0	3.0	EX
Muito Bom	2.0	5.0	MB
Bom	4.0	7.0	BO
Regular	6.0	9.0	RE
Ruim	8.0	10.0	RU
Resultado			
Excelente	0.0	30.0	EX
Ótimo	20.0	50.0	OT
Bom	40.0	70.0	BO
Regular	60.0	90.0	RE
Ruim	80.0	100.0	RU

Uma latência menor entre dois servidores tem um impacto direto no desempenho da comunicação entre dois servidores que trocam mensagens (Gregg, 2010). O conjunto fuzzy *Latência* foi formatado para representar este fato.

A potência de um servidor foi configurada em uma escala de 1 a 10, sendo 1 a melhor potência e 10 a pior. A potência de um servidor é medida em instruções por segundo que o processador que este servidor pode executar.

A Figura 1 ilustra um controlador fuzzy, e um de seus componentes é a base de regras, que mostra como as variáveis se relacionam. Este relacionamento está apresentado na Tabela 6, que fornece todas as regras atingidas. Esta base é formada por duas entradas e uma saída.

Tabela 6 - Base de Regras

Potência \ Latência	**EXCELENTE**	**MUITO BOM**	**BOM**	**REGULAR**	**RUIM**
EXCELENTE	*EXCELENTE*	*OTIMO*	*OTIMO*	*BOM*	*REGULAR*
MUITO BOM	*EXCELENTE*	*OTIMO*	*OTIMO*	*BOM*	*RUIM*
BOM	*EXCELENTE*	*OTIMO*	*BOM*	*REGULAR*	*RUIM*
REGULAR	*OTIMO*	*BOM*	*REGULAR*	*REGULAR*	*RUIM*
RUIM	*BOM*	*REGULAR*	*REGULAR*	*RUIM*	*RUIM*

Quando um servidor recebe uma requisição de recursos e precisa encaminhar esta requisição para outro servidor, aqui denotado servidor Candidato a receber esta requisição, duas informações são utilizadas na validação das proposições fuzzy: a latência entre o servidor que recebeu a requisição e o servidor Candidato, e a potência de processamento, ou poder computacional do servidor Candidato. A validação de uma proposição fuzzy será explicada através de um exemplo.

Supondo uma rede composta por seis servidores que compartilha recursos. A latência entre eles está representada na Tabela 2. Os servidores recebem de clientes solicitações para utilização destes recursos. Em uma situação onde o servidor S1 esteja sem recursos livres e recebe uma solicitação de recursos, o sistema deve procurar por outro servidor para encaminhar esta solicitação.

A inferência fuzzy faz parte do processo de auxilio nesta decisão, e utiliza as duas informações para resolver esta inferência, latência e potência. Para cada outro servidor do grupo de servidores que compartilham recursos, a latência entre S1 e os candidatos a receber esta solicitação é verificada consultando a Tabela 2. O primeiro servidor a ser verificado é N1, e a latência entre S1 e N1 é 89. A potência do servidor N1 é consultada na Tabela 7, verificando-se ter o valor 2.5:

Tabela 7 – Potência dos servidores

Servidor	S1	N1	E1	E2	A1	A2
Potência	5.5	2.5	4.0	1.5	3.0	1.0

Consultando a Tabela 5 Este valor atinge os valores *Excelente* e *Muito Bom* do conjunto *Potência*. Pela Tabela 2, observa-se que a latência entre S1 e N1 tem o valor 89. Este valor atinge as faixas *Bom* e *Regular* do conjunto fuzzy *Latência*. As combinações das variáveis de entrada estão representadas na Tabela 8.

Tabela 8 – Combinações entre Latência e Potência

Se	Latência	e	Potência	Então	Resultado
R01	Excelente		Excelente		Excelente
R02	Excelente		Muito Bom		Excelente
R03	Excelente		Bom		Excelente
R04	Excelente		Regular		Ótimo
R05	Excelente		Ruim		Bom
R06	Muito Bom		Excelente		Ótimo
R07	Muito Bom		Muito Bom		Ótimo
R08	Muito Bom		Bom		Ótimo
R09	Muito Bom		Regular		Bom
R10	Muito Bom		Ruim		Regular
R11	Bom		Excelente		Ótimo
R12	Bom		Muito Bom		Ótimo
R13	Bom		Bom		Bom
R14	Bom		Regular		Regular
R15	Bom		Ruim		Ruim
R16	Regular		Excelente		Bom
R17	Regular		Muito Bom		Bom
R18	Regular		Bom		Regular
R19	Regular		Regular		Regular
R20	Regular		Ruim		Ruim
R21	Ruim		Excelente		Regular
R22	Ruim		Muito Bom		Ruim
R23	Ruim		Bom		Ruim
R24	Ruim		Regular		Ruim
R25	Ruim		Ruim		Ruim

No exemplo dado, são quatro proposições Fuzzy atingidas:

- R11: Se Latência é *Bom* e Potência é *Excelente* então Resultado é *Ótimo*
- R12: Se Latência é *Bom* e Potência é *Muito bom* então Resultado é *Ótimo*
- R16: Se Latência é *Regular* e Potência é *Excelente* então Resultado é *Bom*
- R17: Se Latência é *Regular* e Potência é *Muito bom* então Resultado é *Bom*

Bom, Excelente, Regular, *Muito bom* e *Ótimo* são números fuzzy triangulares, apresentados pela Figura 13:

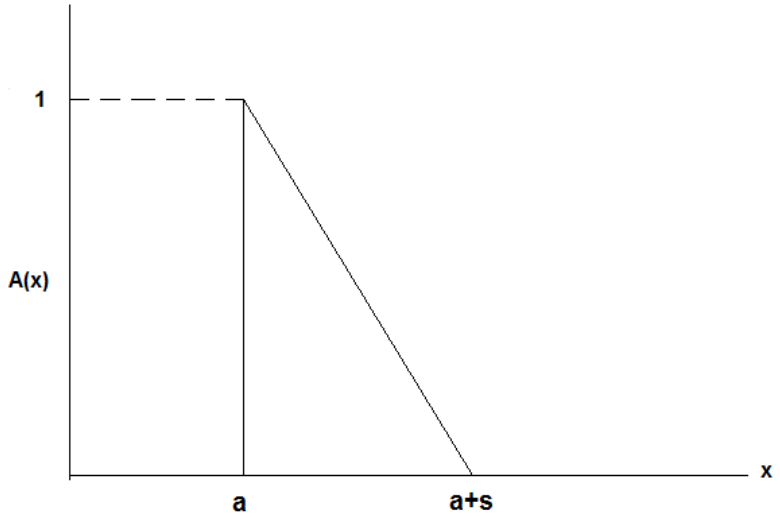

Figura 13 – Número fuzzy triangular (elaborado pelo autor, 2013)

Os valores de Início e Fim podem ser obtidos a partir da Tabela 5. Observa-se que o triângulo escolhido é escaleno, porque o início de cada subconjunto fuzzy neste estudo pertence 100% ao conjunto fuzzy. Por exemplo, uma latência com valor 0 é 100% excelente, de acordo com a teoria fuzzy.

O grau de pertinência é calculado pela função representada pela fórmula (1):

$$A(x) = \begin{cases} 1 - \left|\frac{x-a}{s}\right| & Se\ a \leq x \leq a+s \\ 0 & caso\ contrário \end{cases} \qquad (1)$$

A(x) é o grau de pertinência de um membro de um subconjunto. *A* partir das quatro proposições anteriores tem-se:

- Latência é Bom: A(89)=1-|(89-40)/60|=0,18
- Latência é Regular: A(89)=1-|(89-80)|/70|=0,87
- Potência é Excelente: A(2.5)=1-|(2.5-1.0)/2.0|=0.25
- Potência é Muito bom: A(2.5)=1-|(2.5-2.0)/3.0|=0.83

As quatro proposições fuzzy ficam:

- R11: Se Latência é *0,37* e Potência é *0,25* então Resultado é *Ótimo*

- R12: Se Latência é *0,37* e Potência é *0,83* então Resultado é *Ótimo*
- R16: Se Latência é *0,26* e Potência é *0,25* então Resultado é *Bom*
- R17: Se Latência é *0,26* e Potência é *0,83* então Resultado é *Bom*

O resultado de cada proposição é obtido utilizando-se o método de inferência de Mandani (Nguyen, 2000) que combina os graus de pertinência atingidos nas proposições fuzzy pelo valor mínimo. Obtém-se então para cada proposição:

- R11=mínimo(0,18; 0,25) = 0,18
- R12=mínimo(0,18; 0,83) = 0,18
- R16=mínimo(0,87; 0,25) = 0,25
- R17=mínimo(0,87; 0,83) = 0,83

Cada um destes valores é projetado na variável de saída, de modo que só serão considerados os valores de pertinência que forem menores ou iguais ao valor mínimo obtido. Portanto, em R11 temos o resultado Ótimo em 0,25, ou tem o grau de pertinência ao conjunto de 0,25 (pela teoria fuzzy um elemento que não pertence ao conjunto e tem grau de pertinência igual a 0 ou pertence totalmente ao conjunto e tem grau de pertinência igual a 1). O mesmo se aplica para as outras três proposições, obtendo-se os valores:

- R11: Ótimo em 0,18
- R12: Ótimo em 0,18
- R16: Bom em 0,25
- R17: Bom em 0,83

A Figura 14 ilustra o resultado das proposições R11, R12, R16 e R17.

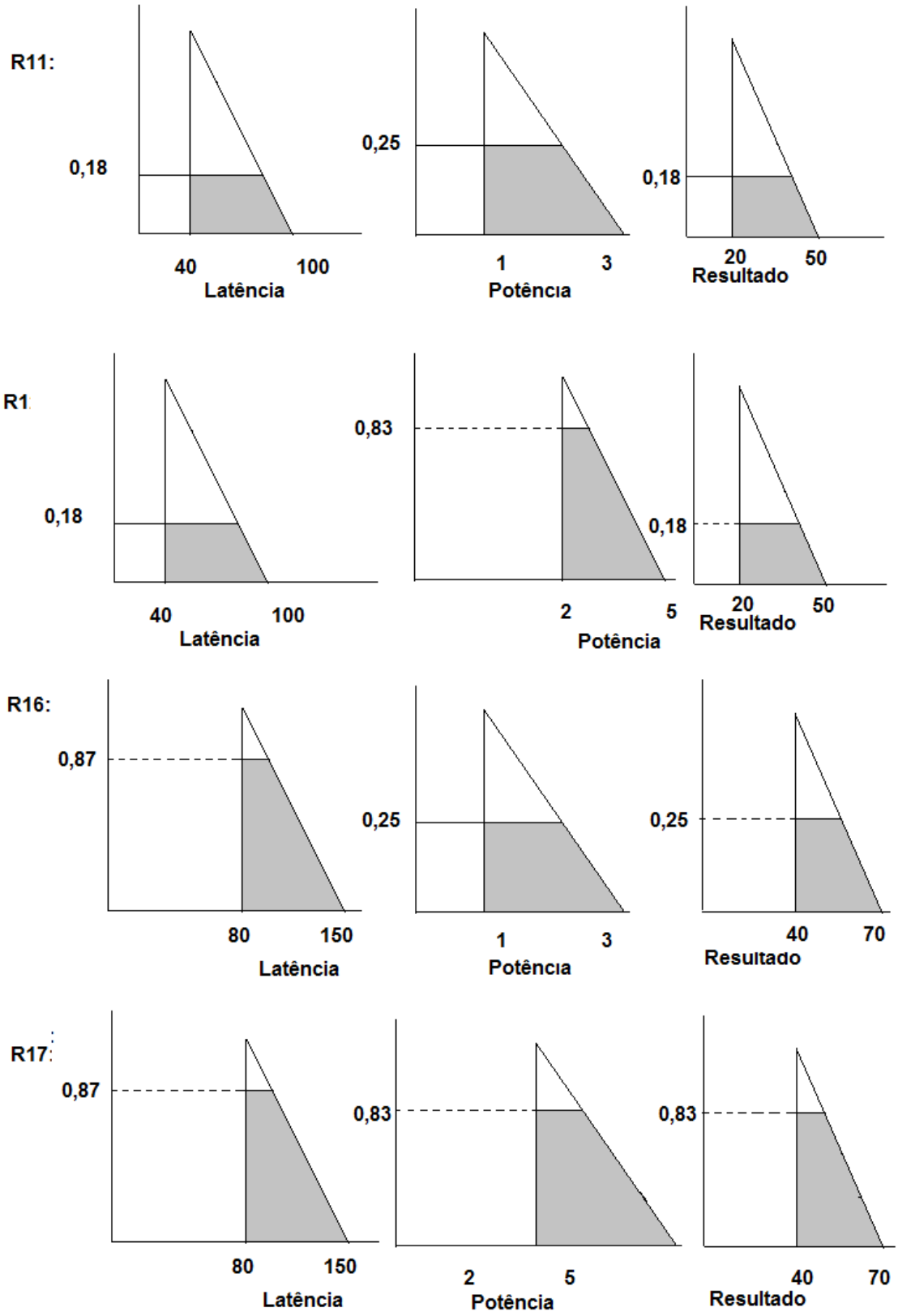

Figura 14 – Resultado das inferências fuzzy atingidas (elaborado pelo autor, 2013)

Combinando as saídas das quatro proposições ativadas, obtém-se a saída geral pelo método de Mandani, sem defuzificação, ilustrada da Figura 15:

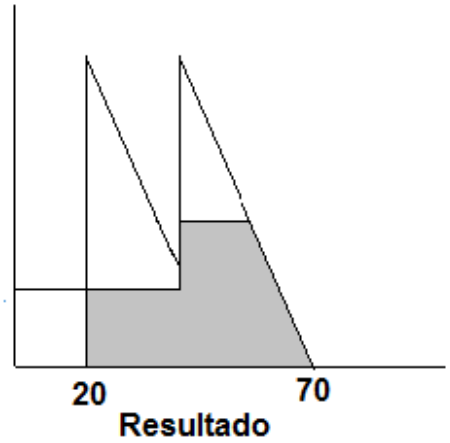

Figura 15 – Saída geral sem defuzificação (elaborado pelo autor, 2013)

A defuzificação é necessária para se obter um número que melhor represente o conjunto. Neste trabalho, o método de defuzificação escolhido foi o do centro de gravidade, que é o método mais utilizado em sistemas com domínio discreto. Este método é definido pela fórmula (2):

$$P = \frac{\int_R xA(x)dx}{\int_R A(x)dx} \qquad (2)$$

Pela Fórmula 2 calcula-se o valor fuzzy entre o servidor S1 e o servidor N1. O processo se repete para cada servidor que tem recursos que podem atender a solicitação. O melhor valor fuzzy indica para qual servidor deve ser encaminhada a solicitação de recursos.

O seguinte pseudocódigo explica esta rotina.

```
Rotina Fuzzy(latência, numero de recursos)
{
     Definir as variáveis de entrada e saída;
     Definir os conjuntos fuzzy que compõe o domínio destas
        Variáveis
     Construir a base de regras que definem a relação destes
```

```
        Conjuntos
      Converter e numero de recursos em números fuzzy;
      Validar as proposições fuzzy
      Defuzificar
Retornar valor defuzificado
}
```

Na próxima seção são feitas considerações para demonstrar que o algoritmo apresentado nesta tese garante a não ocorrência de *deadlock* e *starvation*.

4.7 Considerações sobre *deadlock* e *starvation*

Esta seção apresenta uma demonstração, baseada em uma metodologia apresentada em (Gibilisco, 2005) e (Miles, 2006) da não ocorrência de *deadlock* e *starvation* com a utilização do algoritmo apresentado neste trabalho.

Definição 1: Dado um grupo de servidores S_0, S_1, ..., S_k, este grupo forma o conjunto T dado por:

$$T = \{S_0, S_1, \dots, S_k\}$$

sendo *k+1* o número de servidores deste grupo.

Definição 2: Cada servidor possui m_i recursos compartilhados entre os servidores do conjunto T, sendo m_i maior ou igual a zero.

Definição 3: A definição de "aconteceu antes" é denotada pelo símbolo "→". Se *a* e *b* são eventos do mesmo processo, e *a* ocorre antes de *b*, então *a→b*. Se *a→b* e *b→c* então *a→c*.

Definição 4: Define-se como relógio lógico formado pela tupla <i,r_i> onde i é o valor sequencial do servidor no conjunto T e r é um valor inteiro, com valor inicial 0, e é incrementado por cada processo no recebimento de uma mensagem para atualização de dados entre os servidores.

Definição 5: Uma solicitação de recursos é a requisição de um ou mais recursos feita a um servidor do conjunto T e é indexada pelo relógio lógico do servidor que recebeu esta requisição.

Definição 6: O valor M é a soma dos recursos m de todos os servidores do conjunto T:

$$M = \sum_{i=0}^{k} S_i m_i$$

Definição 7: O vetor MDi[j].MD contém a visão local que o servidor S_i tem do número de recursos disponíveis para compartilhamento que o servidor S_j tem, e o valor D é dado pela soma de todos os recursos disponíveis em todos os servidores do conjunto T:

$$D = \sum_{0 \le i < j \le k}^{k} MD_i[j].MD$$

Definição 8: O vetor REQ_Q_i[y].requisição armazena as solicitações de recursos recebidas pelo servidor S_i

Definição 9: O campo requisição é uma estrutura com quatro campos: Número de recursos solicitados, quem solicitou estes recursos, relógio lógico no momento que a solicitação foi recebida e flag.

Definição 10: A ordenação das requisições armazenadas localmente em cada servidor se dá na comparação da tupla <i, r_i> com <j, r_j>. <i, r_i> é menor do que <j, r_j> se r_i for menor do que r_j. Caso r_i for igual a r_j, <i, r_i> é menor do que <j, r_j> se i<j. Esta ordenação é dada por:

$$(i, r_i) < (j, r_j) \Leftrightarrow (r_i < r_j) \vee ((r_i = r_j) \wedge (i < j))$$

Definição 11: Diz-se que um relógio lógico <i, r_i> tem prioridade sobre um relógio lógico <j, r_j>.se <i, r_i> for menor do que <j, r_j>.

Definição 12: As requisições do vetor descrito na definição 8 são ordenadas por um relógio lógico, da requisição com maior prioridade para a requisição de menor prioridade.

Definição 13: As requisições da fila REQ_Q são atendidas pela ordem indicada na definição 12.

Definição 14: A utilização de um recurso exige um recurso disponível em qualquer servidor do conjunto T.

Definição 15: Define-se o valor *p* como a quantidade de recursos solicitados a todos os servidores, requisições estas com relógio lógico menor do que uma determinada solicitação enviada ao servidor S_i:

$$p = \sum_{0 \le i \le k} PF_i[y].\text{relógio}$$

Definição 16: O valor P para o servidor S_i é dado pela fórmula:

$$P = \sum_{0 \le k, i \ne k} p_i$$

Proposição 1: O algoritmo garante a não ocorrência de *deadlock*.

Demonstração: Se o servidor S_i quer utilizar *n* recursos, a requisição destes recursos foi indexada pelo relógio lógico <i, r_i>. Para que este atendimento seja possível, o algoritmo verifica se o valor *P* calculado pela definição 16, somado ao valor *n* desta requisição, seja menor ou igual ao valor *D*, calculado pela definição 7, como mostrado na thread ack_PENDENTE na seção 4.3. Como esta situação quebra a segunda condição informada na seção 2.3, isto garante a não ocorrência de *deadlock*.

Proposição 2: Se uma requisição assinalada por um relógio lógico <k,r_k> pode ser atendida, então uma solicitação assinalada pelo relógio lógico <i,r_i> pode ser atendida se <i,r_i> tiver prioridade a <k,r_k>.

Demonstração:

Considerando-se a seguinte situação: No momento que o servidor S_k recebe a requisição assinalada com o relógio lógico <k, r_k>, esta requisição é armazenada na fila REQ_Q conforme a definição 8 e, no momento que o servidor Si recebe a requisição assinalada pelo relógio lógico <i,r_i> esta requisição também é armazenada na fila REQ_Q.

Supondo que o valor de *p* indicado pela fórmula da definição 16 seja insuficiente para atender a requisição <i,r_i>, esta requisição permanece na fila REQ_Q aguardando recursos livres. Se a requisição <k,r_k> pode ser atendida pois o valor de *p* no momento que esta requisição é feita tem valor maior ou igual ao número de recursos solicitados, conforme o enunciado da proposição 2, então, pela definição

13 tem-se que todas as requisições com maior prioridade já foram atendidas. Portanto a requisição <i,r_i> já deve ter sido atendida conforme o enunciado da definição 13.

Portanto, esta situação só é possível se a requisição com relógio lógico <i, r_i> for atendida antes da requisição com relógio lógico <k, r_k>.

Proposição 3: Se o servidor S_i quer utilizar *n* recursos, a seguinte Proposição garante a não ocorrência de deadlock:

$$n + \sum_{i=0}^{k} p_i \leq \sum_{0 \leq i < j \leq k} MD_i[j].MD$$

Demonstração:

Sendo *n* o número de recursos de uma solicitação enviada ao servidor S_i. Para cada servidor S do conjunto T há um valor *p* que representa a quantidade de recursos solicitados cujas solicitações estejam assinaladas um relógio lógico com uma prioridade maior do que a requisição dos *n* recursos ao servidor S_i. Cada uma destas requisições pode ser escrita na forma:

<a, r_a>, <j, r_j>,...,<g, r_g>

Supondo que a requisição de n recursos ao servidor S_i tenha o relógio lógico <i, r_i>. Pela definição 16, temos que <i, r_i> tem prioridade menor do que as requisições correspondentes ao valor P. Pela Proposição 2, temos que, para requisição com relógio lógico <i, r_i> ser atendida, todas as outras requisições com prioridade maior devem ser atendidas antes. Se, após estas requisições serem atendidas, temos que para a requisição <i, r_i> poder ser atendida, pela Proposição 1 deve obrigatoriamente haver recursos para atender as requisições dos n recursos da solicitação enviada ao servidor S_i com relógio lógico <i, r_i>.

Proposição 4: Ao solicitar *n* recursos a um determinado servidor Si, esta solicitação será atendida, e em um tempo finito.

Demonstração:

Considerando uma requisição armazenada no vetor REQ_Q com a maior prioridade para ser atendida. Se há recursos disponíveis, pela Proposição 1 esta requisição será atendida. Caso contrário, esta requisição fica bloqueada pelas solicitações que

estão utilizando os recursos do conjunto M. Quando recursos suficientes forem liberados pelos servidores que forneceram estes recursos ao término de sua utilização, uma mensagem <RELEASE> é enviada a todos os servidores do conjunto T, até que a Proposição 1 seja satisfeita para esta requisição.

5 Implementação, Testes e Resultados Obtidos

A implementação do algoritmo apresentado nesta tese foi feita no SimGrid (Casanova, 2008) com o objetivo de verificar a eficiência do mesmo comparado a outros algoritmos de uso comum.

SimGrid é um projeto desenvolvido pela Universidade do Havaí, e das Universidades de Nancy e Grenoble, na França, que possibilita o desenvolvimento de projetos para computação em grade com programação distribuída. Dentre os seus módulos, foi utilizado o GRAS (Grid Reality And Simulation), que é um *framework* para o desenvolvimento de aplicações distribuídas. Este módulo provê uma API para o desenvolvimento de aplicações para serem executadas em plataformas heterogêneas.

As principais funções do GRAS são:

- Sockets: Permite a criação de um canal de comunicação entre processos
- Mensagens: A troca de informações, via sockets, é por troca de mensagens
- Virtualização: É a facilidade de executar o programa como uma simulação ou no mundo real

Um projeto GRAS é constituído basicamente por três arquivos: O programa fonte na linguagem C, que contém o código do programa, um arquivo com a descrição da topologia de rede que compõe o sistema, e um arquivo que descreve a execução do processo de simulação.No Apêndice 1 encontra-se uma descrição destes arquivos.

O SimGrid simula um ambiente que se aproxima do mundo real, nas áreas de rede e processamento distribuído, e tem sido utilizado por pesquisadores em centros de pesquisa e universidades, para prova de conceito (Quinson, 2012). Este fato foi a motivação para utilização deste simulador para verificação do desempenho do algoritmo proposto nesta tese.

5.1 Objetivos dos Testes

A repetida utilização do algoritmo aplicado em um programa desenvolvido no SimGrid teve como objetivo gerar dados que possibilitaram a análise do desempenho da proposta desta tese, em comparação com outras soluções existentes, utilizadas em programas paralelos e distribuídos.

O principal objetivo do trabalho desenvolvido nesta tese é de melhorar o tempo de resposta no atendimento de submissões de aplicações paralelas e sequenciais em um ambiente com múltiplos servidores.

Os pontos que são relevantes na proposta e que são avaliados nos testes são:

- A proposta é aplicada em um ambiente com múltiplos servidores.
- A submissão e o escalonamento são distribuídos.
- Em uma submissão, podem ser solicitados múltiplos recursos.
- A seleção aplicada no escalonamento considera o número de recursos livres em cada nó, ou servidor, e a capacidade de processamento em termos de instruções por segundo que o processador pode executar, e a distância entre o nó de submissão e os demais servidores.

5.2 Plano de testes e análise dos resultados

Os testes propiciaram a geração dos dados para análise de desempenho, comparativas entre a proposta desta tese e algumas outras soluções, em cenários com diversidade na demanda de submissões, no número de servidores, nas distâncias entre servidores e nas capacidades de processamento dos recursos. Este plano foi estruturado em três grupos:

- Análise entre o escalonamento distribuído, que é utilizado na proposta desta tese, e atendimento local em cada servidor.
- Análise entre o escalonamento distribuído desta proposta e escalonamento centralizado.
- Análise da estratégia de seleção de recursos: latência entre os servidores e potência de processamento de cada servidor.

O plano de testes proposto teve a seguinte sequência:

- Escalonamento Distribuído proposto por esta tese comparado a atendimento local, com recursos homogêneos e heterogêneos.
- Escalonamento distribuído proposto por esta tese comparado a outras soluções existentes.
- Escalonamento centralizado, sem e com congestionamento nos servidores de processamento.
- Testes variando a estratégia de seleção de recursos.

Nestes testes os seguintes conceitos são utilizados: recursos homogêneos e heterogêneos, servidores com ou sem congestionamento, tempo médio por requisição.

Considera-se um recurso homogêneo quando todos os servidores possuem a mesma capacidade de processamento. A capacidade de processamento varia entre 1 e 10, sendo 1 o servidor com maior capacidade de processamento, no sentido de poder executar instruções por segundo. O servidor com valor 10 é o que possui menor valor de processamento.

Recursos heterogêneos indica a utilização de uma configuração de servidores onde a capacidade de processamento, em cada servidor, varia. Neste modelo procura-se simular o ambiente onde se tem servidores com diferentes processadores.

O congestionamento de um servidor é caracterizado quando todos os recursos para processamento estejam em utilização, e o sistema local de requisições de recursos tenham solicitações ainda não atendidas. Afirma-se que o servidor está congestionado quando uma requisição de recursos é encaminhada a este servidor e ele não pode executar a solicitação por falta de recursos, nele e em todos os outros servidores que compartilham recursos distribuídos.

Tempo médio por requisição é calculado pela divisão da soma de cada tempo que o sistema leva para completar a execução de uma requisição pelo número total de requisições feitas.

Em cada gráfico foi incluído o valor de Desvio Padrão, variando entre 0,00 e 0,50. No total foram feitos 10 testes para cada situação analisada, que geraram os dados para o cálculo do Desvio Padrão.

A configuração da rede foi composta por 9 servidores com 8 recursos cada, 30 requisições foram feitas para cada servidor, cada requisição levou 2 segundos para seu atendimento, com um fluxo de envio de requisições de 240 RPM. Cada requisição solicitava 6 recursos.

São utilizados três algoritmos nos testes: *Fuzzy*, *RR* e *SL*. *Fuzzy* representa o algoritmo desenvolvido neste trabalho. *RR* representa o algoritmo *Round Robin* e *SL* indica a utilização do algoritmo *Small Latency*.

A funcionalidade dos algoritmos *RR* e *SL* indicam a ordem de escolha de um servidor a enviar uma requisição, na incapacidade do servidor que recebeu a requisição de atender a solicitação.

Em cada servidor a seleção do servidor para atender uma requisição é feita uma busca local em anel, ou seja, se em sua requisição mais recente foi selecionado o servidor i, na próxima requisição a seleção será iniciada pelo servidor i+1.

Para o funcionamento do algoritmo *RR* cada servidor recebe um número entre 0 e *k*, sendo *k*+1 o número de servidores que compõe o conjunto T, conforme descrito na definição 1 da seção 4.7. Uma variável global RR é iniciada com o valor 0 no início da execução deste modelo. O seguinte pseudocódigo ilustra seu funcionamento:

```
Algoritmo Round-Robin
{
     contador = 0;
     RR1 = -1;
     enquanto contador < k
     {
          Se Servidor RR pode atender a solicitação então
               RR1 = RR;
               contador = k;
          fim se
          Se RR < k então
               RR = RR + 1;
          Senão
               RR = 0;
          Fim se
          contador = contador + 1;
     }
     Retorne RR1;
}
```

Se o valor de retorno RR1 for igual a -1 a solicitação é armazenada na fila local e aguarda a liberação de recursos. Caso contrário, RR1 representa o valor de qual servidor, entre 0 e *k*, deve atender a solicitação.

O funcionamento do algoritmo SL é semelhante ao algoritmo RR. Para o funcionamento do algoritmo SL é criado um conjunto *S* baseado no conjunto *T* onde o primeiro elemento do conjunto *S* é o servidor que possui a menor latência entre ele e o servidor que recebeu uma solicitação de recursos e não pode atender esta solicitação. O segundo elemento do conjunto *S* é o servidor que possui a segunda

menor latência entre ele e o servidor que recebeu a solicitação. Isto se repete até o último servidor do conjunto T.

Uma variável global *SL* é iniciada com o valor 0 no início da execução deste modelo, indicando o primeiro servidor do conjunto *S*. O seguinte pseudocódigo ilustra seu funcionamento:

```
Algoritmo Small-Latency
{
     contador = 0;
     SL1 = -1;
     enquanto contador < k
     {
          Se Servidor SL pode atender a solicitação então
               SL1 = SL;
               contador = k;
          fim se
          Se SL < k então
               SL = SL + 1;
          Senão
               SL = 0;
          Fim se
          contador = contador + 1;
     }
     Retorne SL1;
}
```

O valor de retorno SL1 é o valor de qual servidor, do conjunto *S*, deve atender a solicitação.

Uma diferença entre os algoritmos *Fuzzy*, *RR* e *SL* é que, quando uma requisição solicita 6 recursos e o servidor que recebe esta solicitação possui um valor menor do que 6 recursos livres, os algoritmos *RR* e *SL* imediatamente procuram por um servidor que possa atender integralmente esta requisição. Caso nenhum servidor possa atender integralmente, a requisição é armazenada na fila de pedidos e fica aguardando a liberação de recursos para efetivar este atendimento. Conforme descrito neste trabalho, o algoritmo *Fuzzy* procura a melhor composição de servidores com recursos livres para atender totalmente a requisição, caso nenhum servidor possua recursos em quantidade suficiente para antender totalmente a solicitação de recursos.

Nas próximas seções descrevem-se os resultados destes testes.

5.2.1 Escalonamento distribuído proposto e atendimento local por cada servidor

Este teste foi executado para se observar o desempenho da solução proposta por esta tese, onde cada servidor executa uma cópia do algoritmo, encaminhando as solicitações que não podem ser atendidas localmente, com uma solução onde os servidores recebem as solicitações e procuram atender localmente, sem encaminhar as solicitações que não são capazes de resolver.

O objetivo deste teste é de verificar a vantagem ao utilizar um ambiente distribuído, como a proposta desta tese, em relação a outra configuração onde os servidores não encaminham requisições. O resultado obtido está apresentado na Figura 16.

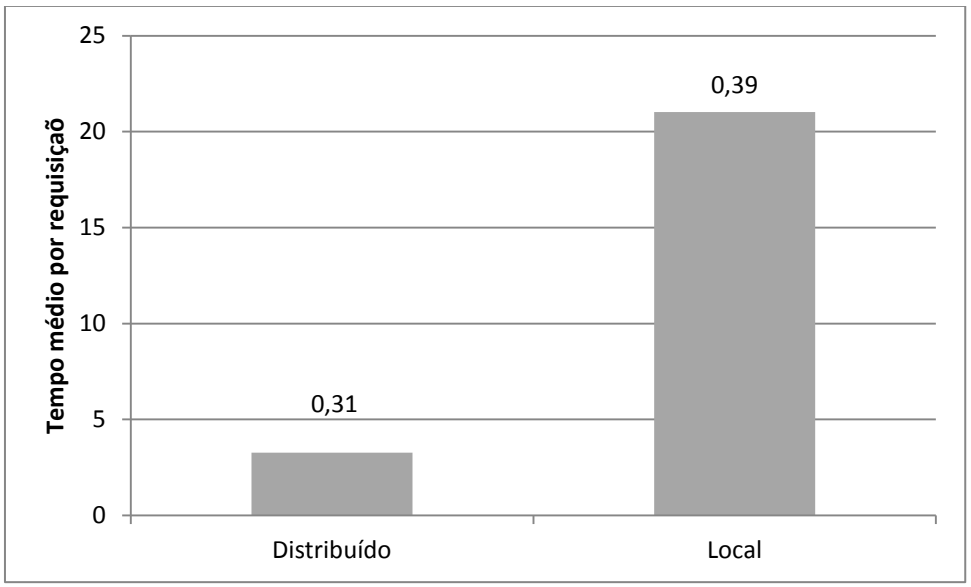

Figura 16 – Escalonamento Distribuído X Atendimento local (elaborado pelo autor, 2013)

O valor de desvio padrão para o ambiente distribuído é 0,31, e para o ambiente local é 0,39.

Neste teste foi considerada a capacidade dos recursos homogêneos para taxa de requisições fixa, sem congestionamento para cada servidor.

Observa-se que, mesmo com recursos homogêneos, a proposta desta tese tem Tempo médio por requisição melhor em relação ao modelo onde o atendimento de solicitações é feita localmente, sem repasse de solicitações a outros servidores, mostrando que é vantajoso utilizar o algoritmo proposto nesta tese, neste cenário,

devido ao fato de que a solução proposta por esta tese aproveita melhor os recursos de todos os servidores, encaminhando as requisições a outros servidores, na incapacidade atender a alguma solicitação, devido a falta de recursos disponíveis localmente.

O mesmo teste foi executado em um ambiente com recursos heterogêneos, e o resultado é apresentado na Figura 17.

Observa-se que, quando os servidores tem poder de processamento heterogêneo, o algoritmo proposto nesta tese tem desempenho melhor do que quando os recursos são homogêneos.

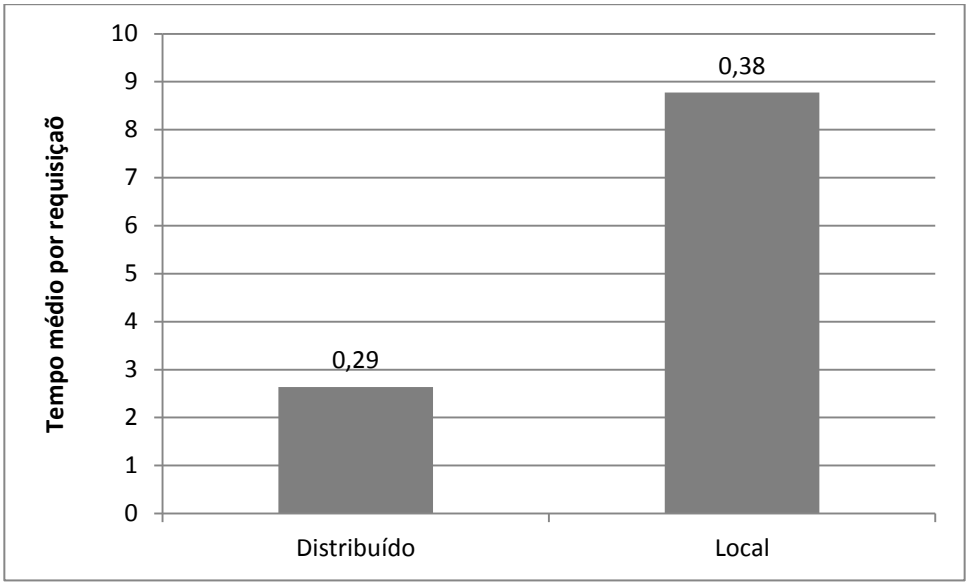

Figura 17 – Escalonamento Distribuído X Atendimento local (elaborado pelo autor, 2013)

Isto deve-se ao fato de que, no algoritmo proposto, como parte da estratégia de decisão na rotina Fuzzy leva-se em conta de encaminhar, preferencialmente, para os servidores com maior poder de processamento as tarefas que não podem ser resolvidas localmente. Com isto o algoritmo procura utilizar os melhores servidores em termos de capacidade de processamento.

5.2.2 Análise entre escalonamento distribuído proposto e escalonamento centralizado

O objetivo dos testes é comparar a solução proposta por esta tese em um ambiente distribuído, onde cada servidor executa uma cópia do algoritmo, e a solução centralizada, onde um único servidor recebe todas as requisições e decide para qual servidor a requisição vai ser encaminhada.

Foram feitos testes em cenários sem congestionamento e com congestionamento.

Em um primeiro conjunto de testes, foi configurado um ambiente sem congestionamento, aplicando-se em um teste o algoritmo de escalonamento distribuído com seleção utilizando a estratégia com lógica fuzzy (fuzzy) e em um segundo e terceiro testes seleções utilizando a estratégia round robin (RR) e menor latência (SL), respectivamente. O resultado está apresentado na Figura 18.

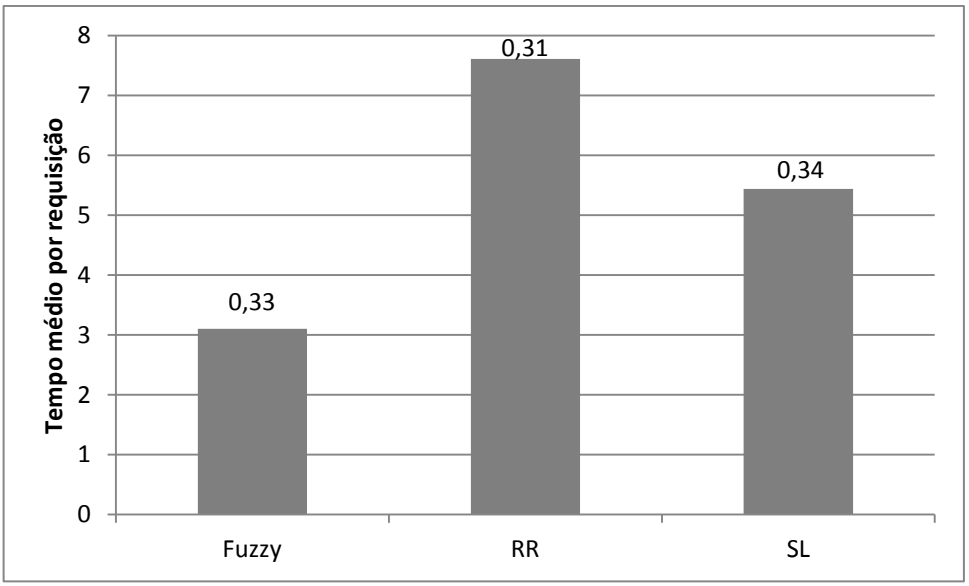

Figura 18 – Comparação entre o algoritmo proposto por esta tese e duas soluções existentes – Modelo Distribuído (elaborado pelo autor, 2013)

Em um segundo conjunto de testes, considerando o mesmo cenário, apenas acrescentando-se um décimo servidor, cuja função é receber as requisições, em um total de 270, e realizar o escalonamento, aplicando-se um algoritmo de escalonamento centralizado com as estratégias de seleção utilizadas no primeiro conjunto de testes. O resultado está apresentado na Figura 19.

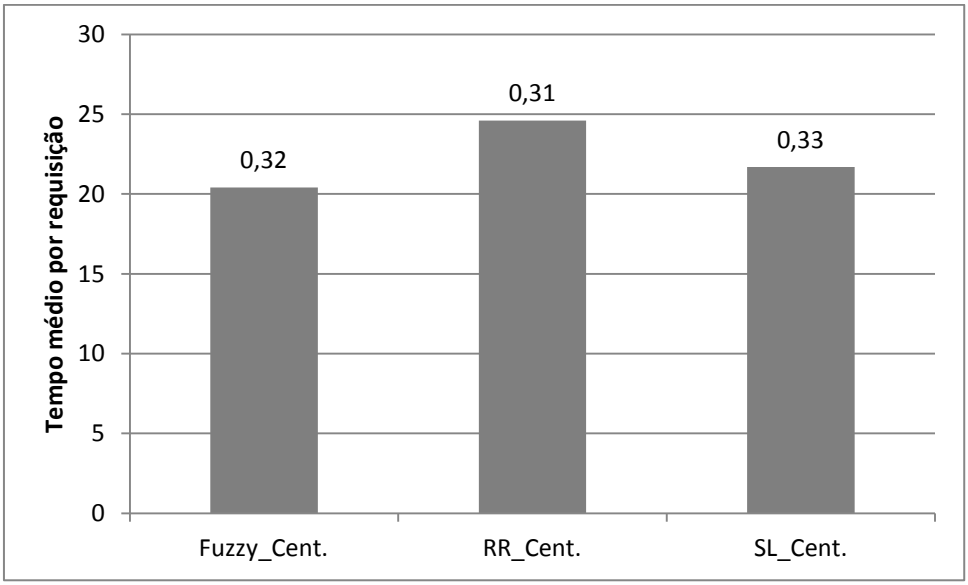

Figura 19 – Solução centralizada proposta por esta tese comparado a duas outras soluções existentes – Modelo Centralizado (elaborado pelo autor, 2013)

Neste cenário, 9 servidores compõe o ambiente de processamento de requisições, e um décimo servidor recebe 270 requisições e executa o algoritmo, decidindo qual dos 9 servidores vai executar a solicitação. O processamento de cada solicitação foi configurada para 2 segundos cada atendimento.

O objetivo deste teste é o de comparar a solução proposta por esta tese, aqui representada por Fuzzy_Cent., com outras duas soluções. Mesmo tendo um único servidor recebendo todas as solicitações e decidindo para qual servidor vai ser encaminhada a requisição, a nossa proposta teve um valor de Tempo médio por requisição menor do que as outras duas soluções.

Em um terceiro conjunto de testes, foram realizados testes aplicando-se o algoritmo de escalonamento distribuído proposto, utilizando a estratégia de seleção fuzzy, em um ambiente variando-se o fluxo de requisições, mas configurando-se um cenário sem congestionamento. Em um cenário similar, adicionando-se o décimo servidor, foram realizados testes aplicando-se o algoritmo de escalonamento centralizado. O resultado é mostrado na Figura 20.

Neste teste, o objetivo é comparar a solução proposta por esta tese em um ambiente distribuído, onde cada servidor executa uma cópia do algoritmo, e a solução centralizada, onde um único servidor, o décimo, recebe todas as requisições e decide para qual servidor a requisição vai ser encaminhada.

O resultado deste teste está apresentado na Figura 20.

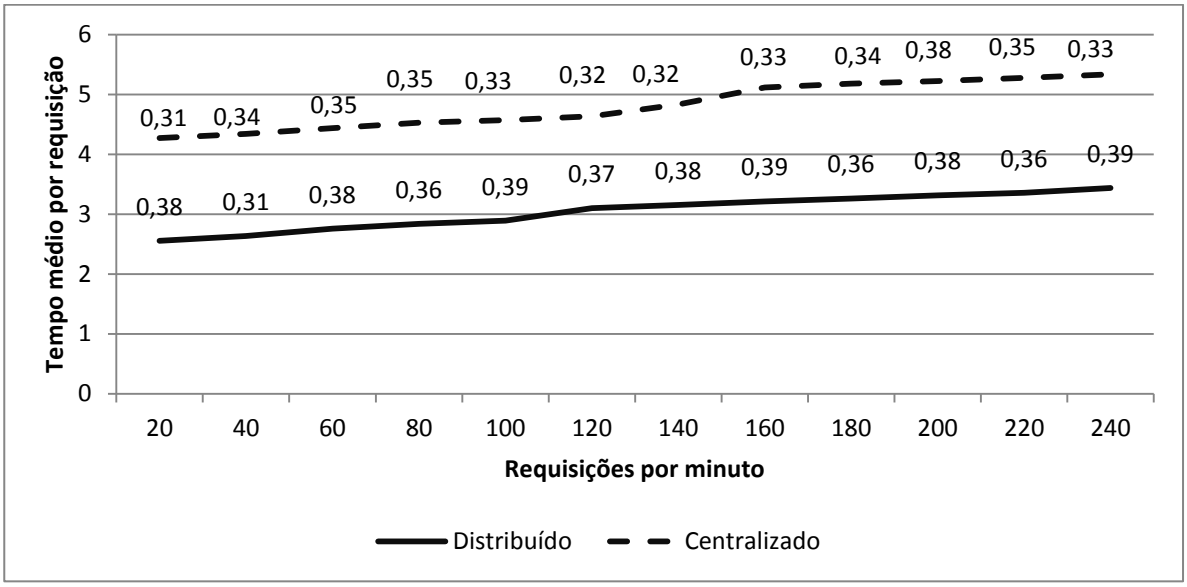

Figura 20 – Comparação entre a solução distribuída e centralizada (elaborado pelo autor, 2013)

Os valores que acompanham as linhas representam o Desvio Padrão para cada ponto do eixo X. Ilustrando, para 20 requisições por minuto, tem-se o Desvio Padrão de 0,38 para o modelo Distribuído e o Desvio Padrão de 0,31 para o modelo Centralizado.

Observa-se, analisando os resultados apresentados, considerando-se um cenário sem congestionamento, a vantagem de se aplicar, em um ambiente de servidores distribuídos, a submissão de requisições distribuídas e um algoritmo de escalonamento distribuído comparado a um modelo centralizado, com a utilização de qualquer uma das estratégias de seleção consideradas nos testes.

O algoritmo proposto por esta tese aplicado em um ambiente distribuído tem um desempenho melhor do que o centralizado, mostrando a vantagem de se utilizar um sistema distribuído para recepção de solicitações de recursos, em relação ao centralizado, apesar do fato de que a instalação do algoritmo no ambiente centralizado é mais simples do que a adaptação do mesmo algoritmo no ambiente distribuído.

Neste teste, variou-se o número de servidores para observar a escalabilidade do algoritmo proposto por esta tese, e o resultado está representado na Figura 21.

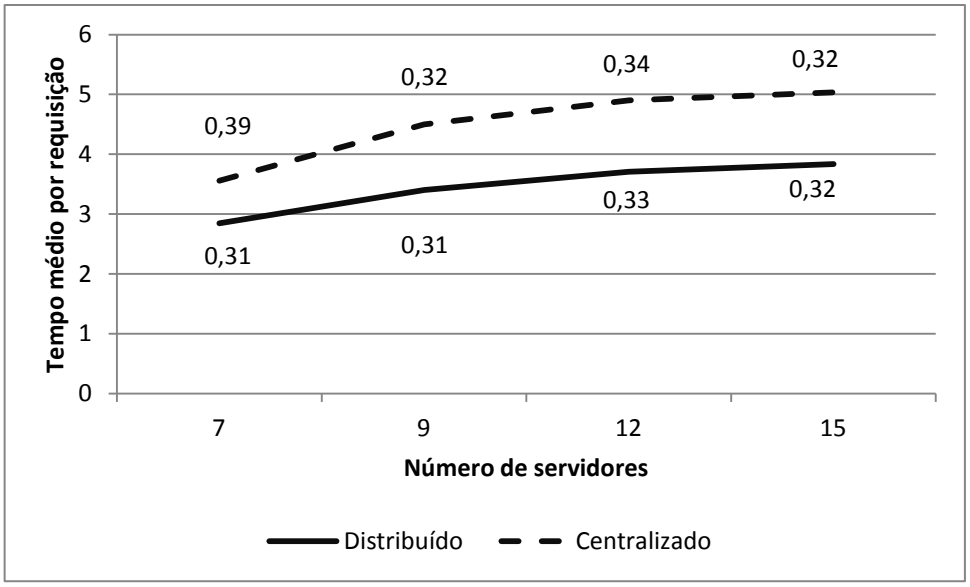

Figura 21 - Distribuído X Centralizado - Variação no número de servidores (elaborado pelo autor, 2013)

Observa-se, na Figura 21, que o aumento do número de servidores mostra uma tendência de um resultado melhor para o algoritmo distribuído pois com 7 servidores a diferença entre o Tempo médio por requisição Centralizado e o Distribuído é menor do que os valores correspondentes para 15 servidores. Isto indica que, quanto maior o número de servidores, um valor melhor de Tempo médio por requisição é o resultado para o algoritmo distribuído.

Isto acontece devido ao fato de que a versão distribuída aproveita melhor os recursos livres em cada nó, tanto no atendimento local como repassando requisições a outros servidores do grupo.

Em um quarto conjunto de testes, foram configurados ambientes com congestionamento. Foram realizados testes em um cenário com 9 servidores, 30 requisições em cada servidor, com fluxos de 60, 120 e 240 requisições por minuto e duração de cada requisição de 20 segundos, aplicando-se a estratégia de seleção Fuzzy e com as estratégias RR e SL. O resultado destes testes está apresentado na Figura 22.

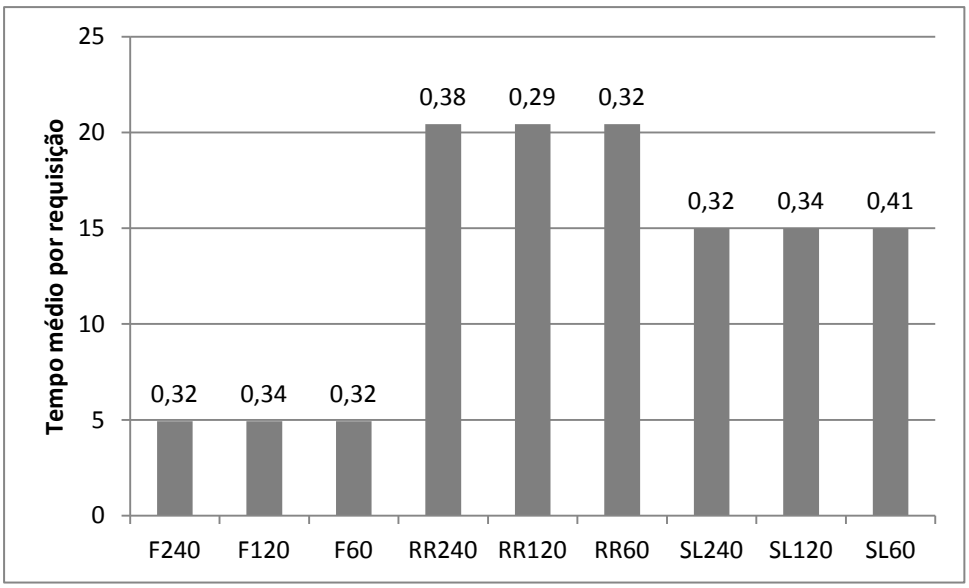

Figura 22 – Algoritmo distribuído proposto com seleção de servidor por fuzzy e outras estratégias de seleção em um cenário com congestionamento - Distribuído (elaborado pelo autor, 2013)

O valor de desvio padrão para F240 é 0,32, ou seja, utilizando o algoritmo Fuzzy com RPM de 240. Os respectivos valores de desvio padrão para os outros testes estão representados acima de cada coluna do gráfico 22.

A ideia deste teste foi de verificar quando os servidores ficam congestionados, o comportamento do sistema em um cenário variando o número de requisições por minuto. Verifica-se que o sistema de filas funciona como uma barreira, pois independente de quantas requisições por minuto o servidor recebe, o tempo de resposta é o mesmo, igualando os tempos de processamento, pois como os recursos não são liberados rapidamente, não importa a intensidade de requisições recebidas pelo servidor.

Em um quinto conjunto de testes, em um cenário similar, adicionando-se o décimo servidor, foram realizados testes solicitando-se um total de 270 requisições neste servidor, e aplicando-se um escalonamento centralizado com as estratégias de seleção utilizadas no quarto conjunto de testes. O resultado deste conjunto de testes está apresentado na Figura 23.

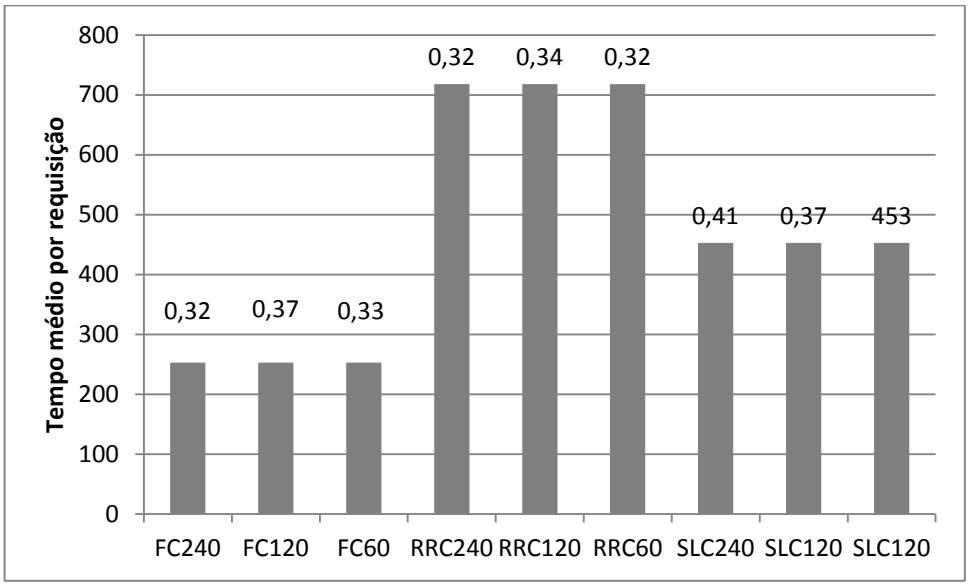

Figura 23 - Verificando quando há congestionamento nos servidores – Solução centralizada (elaborado pelo autor, 2013)

O objetivo deste teste é o de verificar o comportamento do algoritmo de escalonamento centralizado quando os servidores estão congestionados em um ambiente onde apenas um servidor recebe todas as requisições e decide para qual servidor deve ser encaminhada a requisição, aplicando-se os algoritmos de seleção Fuzzy, RR e SL.

Observa-se pelos resultados apresentados nas Figura 22 e Figura 23, que em um cenário de congestionamento, o algoritmo de escalonamento distribuído proposto e o de escalonamento centralizado têm comportamentos similares, independente do número de requisições por segundo que o servidor central recebe solicitações de recursos.

Isto ocorre porque a fila local, quando há congestionamento, funciona como uma barreira, fazendo com que o desempenho geral nas respostas Tempo médio por requisição sejam igualadas. O congestionamento nos servidores faz com que novos pedidos, recebidos pelo servidor, sejam armazenados na fila local de pedidos,

igualando o tempo de resposta, indiferentemente do número de requisições por minuto recebidas pelo servidor.

5.2.3 Estratégia de seleção de recursos

Na estratégia de seleção de recursos proposta, quando um servidor é incapaz de atender uma requisição, procura-se encaminhar esta requisição para o servidor com mais recursos livres. Se mais de um servidor possui o mesmo número de recursos livres, para a tomada de decisão de quem vai receber esta requisição, utiliza-se dois parâmetros, latência entre os servidores e potência de processamento de cada servidor.

Neste primeiro teste, o ambiente foi configurado com recursos homogêneos com número de servidores fixo de 9 servidores, comparando o resultado do Tempo médio por requisição com a taxa de requisições por minuto. O resultado deste teste está apresentado na Figura 24.

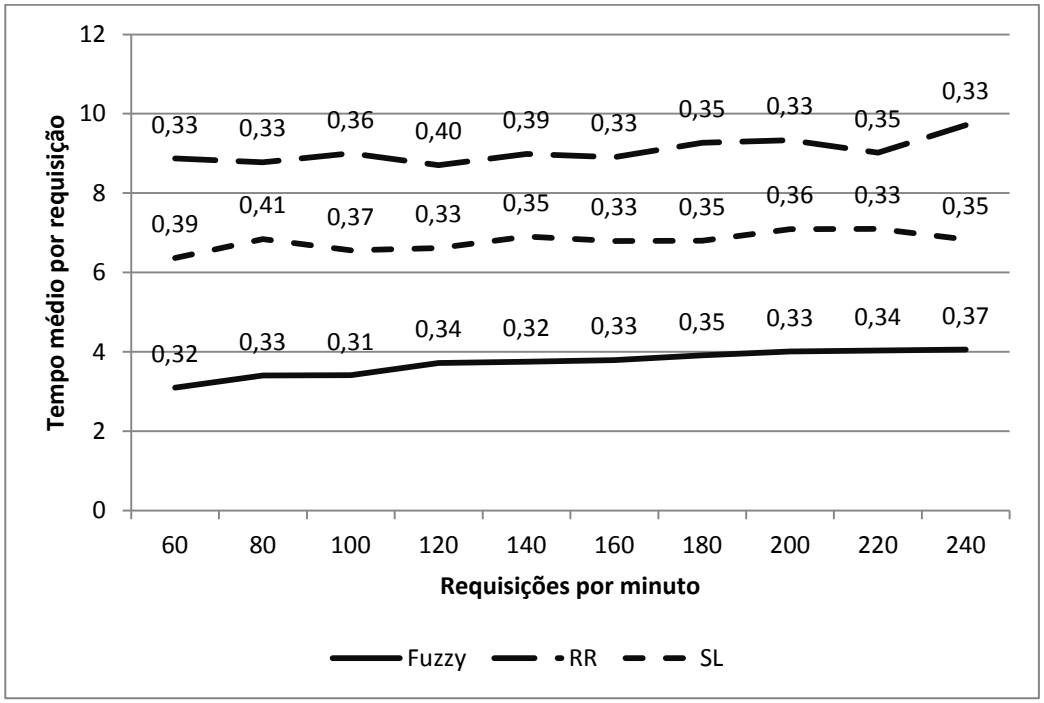

Figura 24 – Tempo médio por requisição X Taxa de requisições: Recursos homogêneos (elaborado pelo autor, 2013)

O objetivo é comparar o desempenho da solução proposta por esta tese, aqui representada por *Fuzzy*, com outras duas soluções existentes. Em um ambiente onde todos os servidores tem a mesma capacidade de processamento observa-se

que o Tempo médio por requisição Fuzzy é melhor do que as duas outras soluções, RR e SL. Isto acontece porque a solução proposta aproveita melhor os recursos ociosos disponíveis em outros servidores.

Neste segundo teste, compara-se o resultado Tempo médio por requisição com a variação no número de servidores. O número de requisições para cada grupo de servidores é fixo e a capacidade de processamento é homogênea. O resultado deste teste está apresentado na Figura 25.

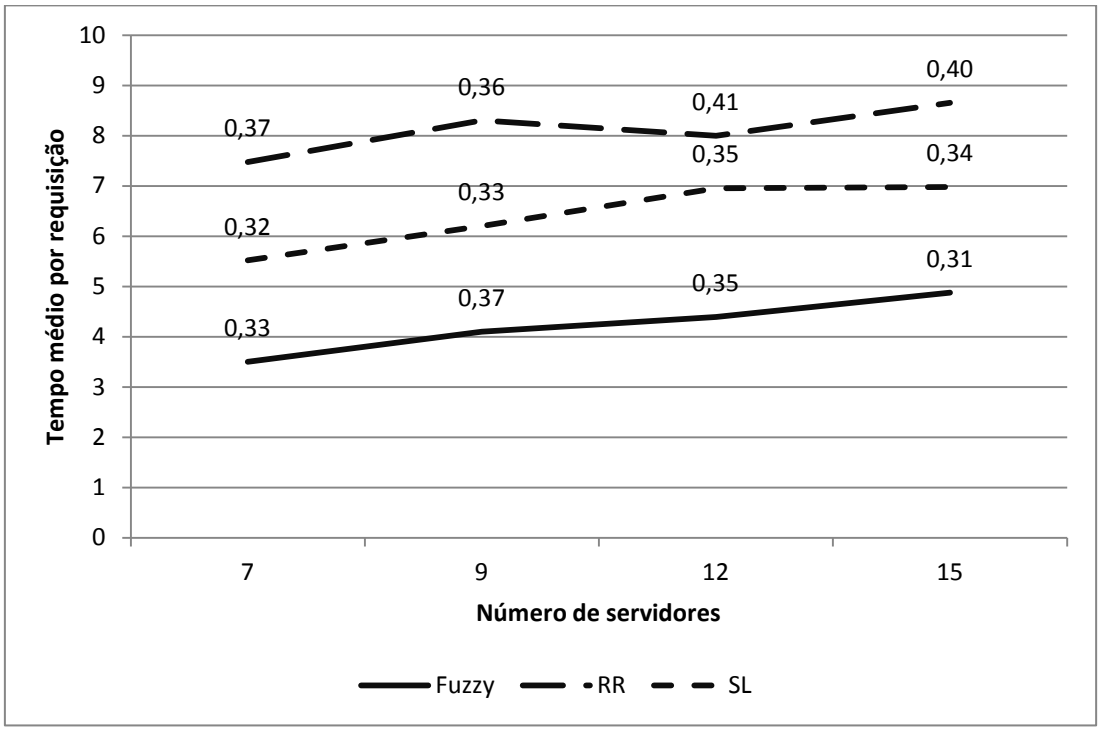

Figura 25 – Tempo médio por requisição X Número de servidores: Recursos homogêneos (elaborado pelo autor, 2013)

Observa-se neste gráfico que, aumentando o número de servidores, o resultado de Tempo médio por requisição para Fuzzy continua melhor do que as outras duas soluções existentes, RR e SL.

Neste terceiro teste o ambiente de servidores foi configurado com recursos com capacidade de processamento heterogêneo. A comparação do Tempo médio por requisição por taxa de requisições por minuto está representada na Figura 26.

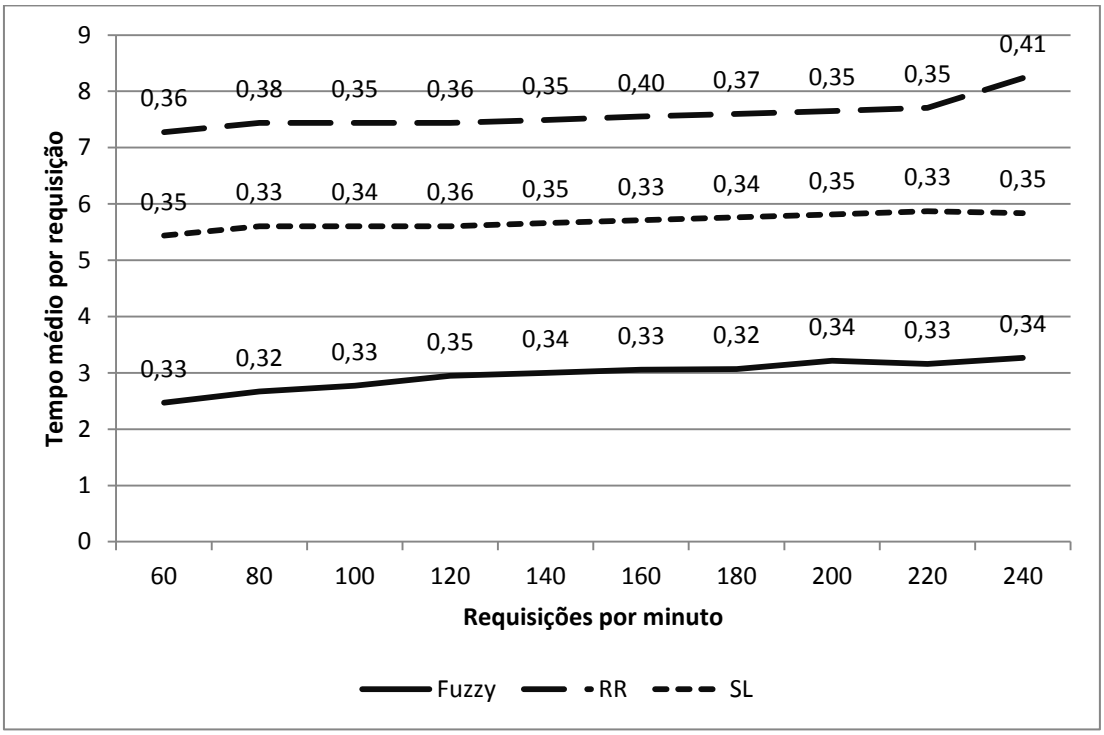

Figura 26 – Tempo médio por requisição X Taxa de requisições: Recursos heterogêneos (elaborado pelo autor, 2013)

O resultado do Tempo médio por requisição para o Fuzzy mostra uma pequena elevação ao se aumentar o número de requisições por minuto. O que é esperado, pois com menor requisições por minuto, a fila local de requisições fica menos sobrecarregada e os servidores podem responder rapidamente as solicitações de recursos.

Neste quarto teste o ambiente foi configurado com o número fixo de 30 requisições por servidor, servidores com recursos heterogêneos, com o objetivo de comparar o Tempo médio por requisição com a variação de número de servidores. O resultado deste teste está apresentado na Figura 27.

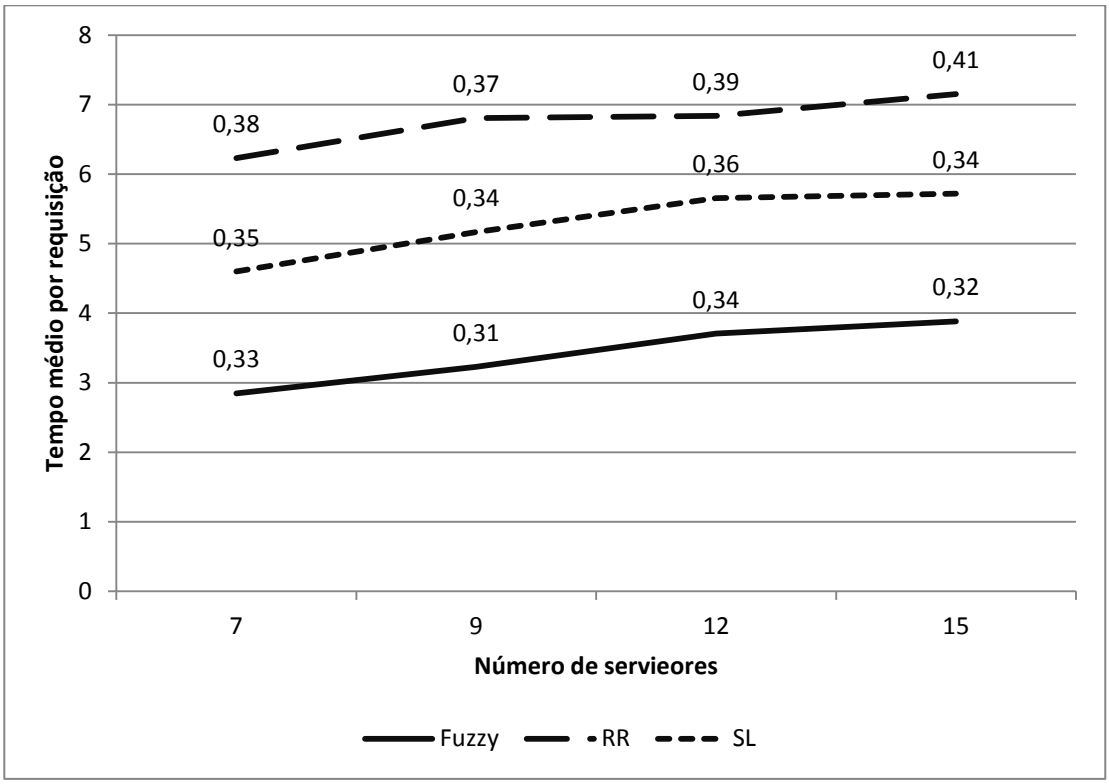

Figura 27 - – Tempo médio por requisição X Número de servidores: Recursos heterogeneos (elaborado pelo autor, 2013)

Observa-se que o desempenho do Tempo médio por requisição para o Fuzzy continua melhor do que as outras duas soluções, RR e SL, mesmo quando há variação no número de servidores.

6. Contribuições e conclusão

São muitos os algoritmos distribuídos para alocação de recursos estudados, mas nenhum apresenta o controle de quem possui os recursos e como estes recursos podem ser utilizados por requisições vindas de outros servidores. Tampouco controlam em qual servidor está sendo executada uma determinada requisição de recursos. Tem-se as seguintes contribuições deste trabalho:

- O algoritmo apresentado neste trabalho permite a utilização de recursos pertencentes a diferentes servidores por uma mesma requisição, e a escolha dos servidores segue um critério que define a melhor combinação de servidores que irão ceder recursos suficientes para atender a solicitação.
- O algoritmo garante que uma requisição será atendida, e em um tempo finito. O controle feito com a combinação de um relógio lógico com um mapa de disponibilidade e a estratégia adotada oferecem um mecanismo de alocação que se mostrou mais eficiente do que estratégias comuns utilizadas atualmente.
- A utilização da lógica fuzzy na estratégia de alocação também é um diferencial com relação a outros trabalhos nesta área. Utilizada no módulo de decisão, a lógica fuzzy mostrou resultados eficientes para o auxílio na escolha do servidor de destino de uma requisição quando mais de um servidor pode receber a solicitação.
- A aplicação do algoritmo se mostrou eficiente quando os servidores estão sobrecarregados com muitas solicitações de recursos. Esta característica torna este trabalho apropriado para o desenvolvimento de soluções cliente-servidor indicando ser apropriado em ambientes com grande número de requisições de recursos.
- Por ser um algoritmo distribuído, a estratégia aplicada no algoritmo permite a inclusão de novos servidores de recursos, possibilitando a criação de um ambiente escalável com relação ao número de recursos oferecidos aos clientes.
- Através da implementação do algoritmo desenvolvido neste trabalho no ambiente SimGrid, foi mostrado que ele é eficiente, em comparação com o algoritmo Round Robin quando os servidores ficam sobrecarregados. A

alocação de recursos distribuídos ainda tem muitos desafios, na área de programação paralela e distribuída.

- Publicação de artigo (Ribacionka, 2012), com o recebimento do prêmio *Best Paper Award* na área de Sistemas Distribuídos.

Como desvantagem, tem-se que, para o funcionamento da estratégia, um grande número de mensagens são trocadas pelos servidores, e quando estes não estão sobrecarregados, os algoritmos tradicionais tem ganho de desempenho comparado com o algoritmo desenvolvido neste trabalho.

Este algoritmo pode ser aplicado em ambientes distribuídos como na computação em grade ou em nuvem, para aplicações que necessitem alocar recursos distribuídos. Como proposta de trabalhos futuros, tem-se:

- Como objetivo comparar o algoritmo deste trabalho com outros algoritmos além do Round Robin e Small Latency.
- Como a troca de mensagens via *socket* no SimGrid é compatível com a programação via API de *socket* no Unix, tem-se como objetivo investigar a aplicação deste algoritmo no mundo real, através da utilização dos computadores instalados no LAHPC da Poli (Laboratório de Arquitetura e de Computação de Alto Desempenho).
- O sistema desenvolvido no SimGrid está de tal forma que pode-se testar outras situações. Na proposta desenvolvida nesta tese utilizou-se a ideia de se utilizar o servidor com maior número de recursos quando este é escolhido para ser utilizado. Mas será interessante pegar o servidor com mais recursos livres? Poderia ser escolhido o segundo maior, para disponibilizar o servidor com mais recursos para outra requisição. Fica a análise de outras configurações do programa desenvolvido nesta tese para posterior investigação, para se verificar se é interessante pegar primeiro o que tem mais recursos, pois o mesmo terá outros recursos, além do processamento, livres, estando menos sobrecarregado em geral. Teoricamente, o servidor tendo menos recursos livres, vai ter mais processos disputando recursos. Por outro lado, é interessante deixar os servidores com mais recursos para os pedidos que vem depois, pois assim se concentra processamento em um só servidor.

Referências Bibliográficas

Arafah, M. A, **Grid computing: a STOPE view**, International Journal of Network Management, 2006

Bahi J. M., Contassot-Vivier S., Giersch A., **Load balancing in dynamic networks by bonded delays asynchronous diffusion**. In: José M. Laginha M. Palma, Michel-Daydé, Osni Marques and João Correia Lopes (Eds). VECPAR 2010. LNCS, vol. 6449, pp. 352--365. Springer, (2010)

Baldoni R., **An O(NM/(M+1)) distributed algorithm for the k-out of-M resources allocation problem**, The 14th International Conference on Distributed Computing Systems, pp.81–88 (1994).

Barros, L. C, Bassanezi, R. C., **Tópicos de Lógica Fuzzy e Biomatemática**, Coleção IMECC Textos Didáticos, Volume 5, 2006

Beumont, Oliver, Eyraud-Dubois, Lionel, Caro Christopher Thraves, Rejeb mejer, **Heterogeneous Resource Allocaton under Degree Constrains**, IEEE on Parallel and Distributed Systems, vol 24, pg. 926-937, 2013

Ben-Ari, M., **Principles of Concurrent and Distributed Programming**, Addison-Wesley, 2006

Berman et al , **The grid: past, present, future**, in Grid Computing: Making the Global Infrastructure a Reality, Wiley, 2003

Breshears, C., **The Art of Concurrency**, O'Reilly, 2009

Casanova, H., Legrand, A., Quinson, M., **SimGrid: A Generic Framework for Large-Scale Distributed Experiments**, 10o. Conference on Computer Modeling and Simulation, UKSIM 2008, pages 126-131.

Celikyilmaz, A, Turksen, I. B., **Modeling Uncertainty with Fuzzy Logic With Recent Theory and Applications**, 2009 Springer-Verlag

Chao-Chin, Wu , **An integrated security-aware job scheduling strategy for large-scale computational grids**, Elsevier: Future Generation Computer Systems 26, 2010

Chen, Haitao, Lu, Yutong, Zhu, Qinghua, **A Power-aware Job Scheduling Algorithm**, International Conference on Cloud Computing and Service Computing, IEEE, pg 8-11, 2012

Coulouris, G., Dollimore, J., Kindberg,T., **Distributed Systems – Concepts and Design**, Pearson, 2001

Cox, E., **The Fuzzy Systems Handbook**, AP Professional, 1999

Ferreira, L, et al, **Introduction to Grid Computing with Globus**, IBM Redbooks, 2003.

Foster, I., et al, **A Distributed Resource Management Architecture**, Proceeding so the International Workshop on Quality of Service, 1999

Foster, I., Kesselman, C., Tuecke, S., **The Anatomy of the Grid**, Proceedings of the 7th International Euro-Par conference Manchester on Parallel Processing, pages 1-4, 2001

Garg, Vijay k., **Elements of Distributed Computing**, Wiley & Sons, Inc., 2002

Gibilisco, S. **Math Proofs Demystified**, McGRAW-HILL, 2005

Gregg, Brendan, **Visualizing system latency**, Communications of the ACM, vol:53 iss: 7 pg:48-54, 2010

Holt, Graham, **Time-Critical Scheduling on a Well Utilised HPC System at ECMWF Using Loadleveler with Resource Reservation**, Lecture Notes in Computer Science Volume 3277 2005, pp 102-124

Jiang, Jehn-Ruey, **Nondominated local coteries for resource allocation in grids and clouds**, Information Processin Letters 111, Elsevier, pg. 379-384, 2011

Jing, Xiao, Zhiyuan Wang, **A Priority base Scheduling Strategy for Virtual Machine Allocation in Cloud Computing Environment**, International Conference on Cloud Computing and Service Computing, IEEE, pg 50-55, 2012

Khazaei, Hamzeh, Misic Jelena, Misic B. Vojislav, Rashwand Saeed, **Analysis of a Pool Management Scheme for Cloud Computing Centers**, IEEE Transactions on Parallel and Distributed Systems, Vol. 24, No. 5, pg. 849-861, 2013

Khuen C. W., Yong C. H., Haron F., **A Framework for Multi-Agent Negotiation System**, International Journal of Information Technology, Vol. 11, No. 4, 2011.

Kirk, D. B, Hwu, W. W., **Programming Massively Parallel Processors - A Hands-on Approach**, Morgan kaufmann, 2010

Klir, George J., Yuan, Bo, **Fuzzy Sets and Fuzzy Logic: Theory and Applications**, Prentice Hall, 1995

Krauter, K, Buyya, R., Maheswaran, M., **A taxonomy and survey of grid resource management systems for distributed computing,** Software: Practice and Experience (SPE), ISSN: 0038-0644, Volume 32, Issue 2, Pages: 135-164, Wiley Press, USA, February 2002

Kurose, J. F., Ross, K. W., **Computer Networking: A Top-Down Approach** , Pearson, 5th Edition, 2009

Lamport, L, **Time, Clocks and the Ordering of Events in a Distributed System**, Communications of the ACM, July 1978, volume 21, number 7.

Leal, Katia et al., **A decentralized model for scheduling idependent task in Federated Grids**, Elsevier: Future Generation Computer Systems 25 (2009) 840-852, 2009

Levesque, J., **High Performance Computing - Programming and Applications**, CRC Press, 2011

Magoulès, F., Pan, J., Tan, Kiat-An, **Introduction to Grid Computing**, CRC Press, 2009.

Maekawa M.,. **A sqrt(N) algorithm for mutual exclusion in decentralized systems**. ACM- Transactions on Computer Systems, 3(2):145{159, May 1985

Martínez, J. A., Almeida, F., Garzón, E. M., Costa, A., Blanco, V., **Adaptive load balancing of iterative computation on heterogeneous nondedicated systems**, Springer Science+Business Media, LLC 2011.

Mileff, P., Nehez, K., **Fuzzy Based Load Balancing for J2EE Applications**, Production Systems and Information Engineering Volume 3 (2006), pp.57-71.

Miles, C. P., Coelho, S. P., **Números - Uma introdução à Matemática**, edusp, 2006

Naimi M., Trehel M.,. Arnold A.. **A log (N) distributed mutual exclusion algorithm based on path reversal**. Journal of Parallel and Distributed Computing, 34(1):1-13, 1996

Nakai, M. A., Madeira, E., Buzato, L. E., **Load Balancing for Internet Distributed Services using Limited Redirection Rates**, Proceedings of the 5th Latin-American Symposium on Dependable Computing. April, 2011.

Nguyen, H. T., Walker, E. A., **A First Course in Fuzzy Logic**, Chapman & Hall / CRC, Second Edition, 2000.

Parhami, B., **Introduction to Parallel Processing - Algorithms and Architectures**, Kluwer Academic Publishers, 2002.

Park, C., Kuhl J. G., **A fuzzy-based distributed load balancing algorithm for large distributed systems**, Proceedings of the Seventh International Workshop on Computer-Aided Software Engineering, IEEE Computer Society, 1995

Paula, N. C., **Um ambiente de monitoramento de recursos e escalonamento cooperativo de aplicações paralelas em grades computacionais**, tese de doutorado, Escola Politécnica, USP, 2009

Quinson, Martin, Rosa Cristian, Thiéry, Christophe, **Parallel Simulation of Peer-to-Peer Systems**, In Proceedings of the 12th IEEE International Symposium on Cluster Computing and the Grid (CCGrid´12), IEEE Computer Society Press, May 2012

Raynal, M.: **A distributed solution to the k-out of-M resources allocation problem**, Institut de Recherche en Informatique Et Systemes Aleatoires, 1991

Reddy V. A., Mittal P., Gupta I., **Fair k mutual exclusion algorithm for peer to peer systems**, in IEEE International Conference on Distributed Computing Systems, 2008, pp. 655–662

Reveliotis, S. A., **Real-Time Management of Resource Allocations Systems**, Springer, 2005

Ribacionka, F., **Sistemas Computacionais Baseados em Lógica Fuzzy**, Dissertação (Mestrado) - Faculdade de Engenharia, Universidade Presbiteriana Mackenzie, 2008

Ribacionka, F, Sato, L. M., Arantes, L., **A Distributed Resource Allocation Algorithm Using Fuzzy Logic**, Proceedings of the IASTED International Conference, Parallel and Distributed Computing and Systems (PDCS 2012), November 12-14, Las Vegas, USA, pp. 46-53, 2012

Ricart G., Agrawala A., **An optimal algorithm for mutual exclusion in Computer Networks**. Communications of the ACM, 24, 1981

Rodamilans, C., **Análise de desempenho de algoritmos de escalonamento de tarefas em grids computacionais usando simuladores**, Dissertação (Mestrado) – Escola Politécnica, Universidade de São Paulo, 2009

Schopf, M. Jennifer, **Ten Actions When Grid Scheduling**, Grid Resource Management: State of the Art and Future Trends, p. 15-23, 2004

Silberschatz, A., Galvin, P. B., Gagne, G. ,**Operating System Concepts**, seventh edition, John Wiley & Sons. Inc, 2005

Sharma, D., Saxena, A. B., **Framework to solve load balancing problem in heterogeneous web servers**, International Journal of Computer Science & Engineering Survey (IJCSES) Vol. 2, No. 1, Feb. 2011

Smith J. F., Rhyne R. D., **A Fuzzy Logic Algorithm for Optimal Allocation of distributed Resources**, Japan Aerospace Exploration Agency, 2000 at http://airex.tksc.jaxa.jp/pl/dr/20010016329/en

Srimani P. K. , Reddy R. L. N., **Another distributed algorithm for multiple entries to a critical section**, Information Processing Letters,vol. 41, no. 1, pp. 51–57, 1992.

Srivasta P., Gupta S., Yadav D. S., **Improving Performance in Load Balancing Problem on the Grid Computing System**, International Journal of Computer Applications (0975 – 8887) Volume 16– No.1, February 2011.

Stevens, W. R., **UNIX Network programming: Interporcess Communications**, Volume 2, 2009

Subramani, V., Kettimuthu, R., Srinivasan, S, Sadayappan, P., **Distributed Job Scheduling on Computational Grids using Multiple Simultaneous Requests**,

Proceedings of 11th IEEE Symposium on High Performance Distributed Computing (HPDC 2002), July 2002

Suzuki I, Kasami T,. **A distributed mutual exclusion algorithm**. ACM Transactions on Computer Systems, 3(4):344-349, 1985.

Thain D., Tannenbaum T., Livny M., **Distributed Computing in Practice: The Condor Experience**, Concurrency and Computation: Practice and Experience, Vol. 17, No. 2-4, pages 323-356, February-April, 2005

Tanenbaum, A. S., **Sistemas Operacionais Modernos**, 3ª. Edição, Pearson Education do Brasil, 2010

Ting, He, Shiyao Che, Hyoil Kim, Lang, Tong, Kang-Won Lee, **Scheduling Parallel Tasks onto Opportunistically Available Cloud Resources**, International Conference on Cloud Computing and Service Computing, IEEE, pg 180-187, 2012

Wang, Wei, Zeng, Guosun, **Bayesian Cognitive Model n Scheduling Algorithm for Data Intensive Computing**, Journal of Grid Computing, Vol 10(1) pg 173-184, 2012

Weissman, J. B., Grimshaw, A. S., **A Federated Model for Scheduling in Wide-Area Systems**, HPDC Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing, 1996

Zadeh, L. A., **Fuzzy sets, Information and Control**, vol. 8, pg. 338-353, 1965.

Zong, Wang Jiang, Qiu, Zheng Sheng, **A New Task Scheduling Algorithm in Hybrid Cloud Environment**, Internationa Conference on Cloud Computing and Service Computing, IEEE, pg. 45-49, 2012

Apêndice 1

Arquivo deployment_alocrec_xml

Este arquivo descreve como o seu programa vai ser iniciado, por qual rotina e em qual máquina virtual, e com quais parâmetros. Nesta simulação são utilizados seis computadores virtuais, cada um deles fazendo o papel de um servidor Web que compõe o grupo que vai compartilhar recursos. Em cada um destes, a rotina inicial é chamada de *vm* e é onde o sistema inicia a execução do programa. Este arquivo está ilustrado na Figura 28.

```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "SimGrid.dtd">
<platform version="3">
  <process host="vm0" function="vm" />
  <process host="vm1" function="vm" />
  <process host="vm2" function="vm" />
  <process host="vm3" function="vm" />
  <process host="vm4" function="vm" />
  <process host="vm5" function="vm" />
</platform>
```

Figura 28 – Arquivo deployment_alocrec.xml

Arquivo platform_alocrec.xml

O arquivo platform_alocrec.xml é utilizado para informar ao simulador sobre os servidores e a relação entre eles. A identificação de cada um é feita pelo *host id* e a capacidade de processamento é informada pelo comando *power*, que para a simulação desta implementação foi utilizado o valor padrão fornecido no tutorial do SIMGRID[5]. O mesmo foi feito para o valor da largura de banda (bandwidth). O valor da latência foi retirado da Tabela 2. A Figura 29 ilustra este arquivo.

```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "SimGrid.dtd">
<platform version="3">
  <host id="S1" power="98095000"/>
  <host id="N1" power="98095000"/>
  <host id="E1" power="98095000"/>
  <host id="E2" power="98095000"/>
  <host id="A1" power="98095000"/>
  <host id="A2" power="98095000"/>
  <link id="01" bandwidth="3430125" latency="0.089"/>
```

[5] http://SimGrid.gforge.inria.fr/SimGrid/3.5/doc/GRAS_tut_tour_setup.html

```
  <link id="02" bandwidth="3430125" latency="0.138"/>
  <link id="03" bandwidth="3430125" latency="0.140"/>
  <link id="04" bandwidth="3430125" latency="0.193"/>
  <link id="05" bandwidth="3430125" latency="0.272"/>
  <link id="12" bandwidth="3430125" latency="0.048"/>
  <link id="13" bandwidth="3430125" latency="0.058"/>
  <link id="14" bandwidth="3430125" latency="0.109"/>
  <link id="15" bandwidth="3430125" latency="0.156"/>
  <link id="23" bandwidth="3430125" latency="0.018"/>
  <link id="24" bandwidth="3430125" latency="0.151"/>
  <link id="25" bandwidth="3430125" latency="0.114"/>
  <link id="34" bandwidth="3430125" latency="0.162"/>
  <link id="35" bandwidth="3430125" latency="0.122"/>
  <link id="45" bandwidth="3430125" latency="0.068"/>
  <route src="S1" dst="N1"><link:ctn id="01"/></route>
  <route src="S1" dst="E1"><link:ctn id="02"/></route>
  <route src="S1" dst="E2"><link:ctn id="03"/></route>
  <route src="S1" dst="A1"><link:ctn id="04"/></route>
  <route src="S1" dst="A2"><link:ctn id="05"/></route>
  <route src="N1" dst="S1"><link:ctn id="01"/></route>
  <route src="N1" dst="E1"><link:ctn id="12"/></route>
  <route src="N1" dst="E2"><link:ctn id="13"/></route>
  <route src="N1" dst="A1"><link:ctn id="14"/></route>
  <route src="N1" dst="A2"><link:ctn id="15"/></route>
  <route src="E1" dst="S1"><link:ctn id="02"/></route>
  <route src="E1" dst="N1"><link:ctn id="12"/></route>
  <route src="E1" dst="E2"><link:ctn id="23"/></route>
  <route src="E1" dst="A1"><link:ctn id="24"/></route>
  <route src="E1" dst="A2"><link:ctn id="25"/></route>
  <route src="E2" dst="S1"><link:ctn id="03"/></route>
  <route src="E2" dst="N1"><link:ctn id="13"/></route>
  <route src="E2" dst="E1"><link:ctn id="23"/></route>
  <route src="E2" dst="A1"><link:ctn id="34"/></route>
  <route src="E2" dst="A2"><link:ctn id="35"/></route>
  <route src="A1" dst="S1"><link:ctn id="04"/></route>
  <route src="A1" dst="N1"><link:ctn id="14"/></route>
  <route src="A1" dst="E1"><link:ctn id="24"/></route>
  <route src="A1" dst="E2"><link:ctn id="34"/></route>
  <route src="A1" dst="A2"><link:ctn id="45"/></route>
  <route src="A2" dst="S1"><link:ctn id="05"/></route>
  <route src="A2" dst="N1"><link:ctn id="15"/></route>
  <route src="A2" dst="E1"><link:ctn id="25"/></route>
  <route src="A2" dst="E2"><link:ctn id="35"/></route>
  <route src="A2" dst="A1"><link:ctn id="45"/></route>
</platform>
```

Figura 29 - platform_alocrec.xml

Esta topologia permite que todos os servidores consigam enviar e receber mensagens de todos os outros, para uma total comunicação entre eles.