FUNDAÇÃO DE ENSINO “EURÍPIDES SOARES DA ROCHA”
CENTRO UNIVERSITÁRIO “EURIPIDES DE MARILIA” - UNIVEM
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

RAPHAEL NEGRISOLI BATISTA

ARQUITETURA DE BALANCEAMENTO DE CARGA EM AMBIENTES PARALELOS DISTRIBUÍDOS UTILIZANDO SISTEMAS MULTIAGENTES

MARÍLIA
2011

RAPHAEL NEGRISOLI BATISTA

ARQUITETURA DE BALANCEAMENTO DE CARGA EM AMBIENTES PARALELOS DISTRIBUÍDOS UTILIZANDO SISTEMAS MULTIAGENTES

Trabalho de Curso apresentado ao Curso de Bacharel em Ciência da Computação da Fundação de Ensino "Eurípides Soares da Rocha", mantenedora do Centro Universitário Eurípides de Marília - UNIVEM, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador:
Prof. Me. MAURÍCIO DUARTE

MARÍLIA
2011

Batista, Raphael Negrisoli

Arquitetura de Balanceamento de carga em ambientes paralelos distribuídos utilizando sistemas multiagentes / Raphael Negrisoli Batista; orientador: Maurício Duarte. Marília, SP: [s.n.], 2011. 80 f.

Trabalho de Curso (Graduação em Ciência da Computação) - Curso de Ciência da Computação, Fundação de Ensino "Eurípides Soares da Rocha", mantenedora do Centro Universitário Eurípides de Marília - UNIVEM, Marília, 2011.

1. Sistemas distribuídos 2. Balanceamento de carga 3. Sistemas multiagentes

CDD: 005.4476

CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

TRABALHO DE CONCLUSÃO DE CURSO – AVALIAÇÃO FINAL

Raphael Negrisoli Batista

ARQUITETURA DE BALANCEAMENTO DE CARGA EM AMBIENTES PARALELOS DISTRIBUIDOS UTILIZANDO SISTEMAS MULTIAGENTES

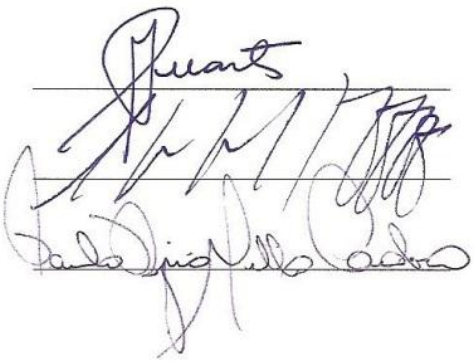
Banca examinadora da monografia apresentada ao Curso de Bacharelado em Ciência da Computação do UNIVEM/F.E.E.S.R., para obtenção do Título de Bacharel em Ciência da Computação.

Nota: 10 (dez)

Orientador: Mauricio Duarte

1°. Examinador: Emerson Alberto Marconato

2°. Examinador: Paulo Rogerio de Mello Cardoso



Marília, 28 de novembro de 2011.

Aos meus pais e ao meu irmão.

A Mairis.

Agradecimentos

"Cheguei aonde cheguei porque me apoiei em ombros fortes."(Isaac Newton)

Agradeço primeiramente a Deus que me deu o dom da vida e a encheu de pessoas tão especiais.

Agradeço aos meus pais José Sergio Batista e Maria Inêz Negrisoli Batista, por tudo o que fazem por mim, sem eles não chegaria até aqui. Tudo o que sou devo a eles.

Ao meu querido irmão Daniel Negrisoli Batista, meu melhor amigo, que sempre me ajudou e me incentivou com sua sabedoria e conhecimento.

A minha namorada, Mairis Porquieres Romero, por tanta compreensão, e por me incentivar nos momentos difíceis que encontrei. "É melhor serem dois do que um, porque se um cair o outro o levanta".

Ao amigo Jonathan Schneider, por sua imensa ajuda na infraestrutura durante momentos difíceis no desenvolvimento deste trabalho. Ao amigo Luis Fernando Martins Carlos Junior, pelo companheirismo durante inúmeros trabalhos desenvolvidos durante a graduação. E ao amigo Denis Renato de Moraes Piazentin, por me incentivar a sempre fazer o melhor de uma forma como só ele sabe, "Você está fazendo alguma coisa errada!", nunca me esquecerei desta frase.

Ao grande mestre Maurício Duarte pela paciência e orientação.

A todos meus amigos de graduação, todos os heróis que conseguiram chegar ao fim desta jornada.

A todos os meus professores de graduação pelos desafios e incentivos a pesquisa científica.

Obrigado.

Raphael Negrisoli Batista

"As pessoas que são loucas o suficiente
para achar que podem mudar o mundo
são aquelas que o mudam."

(Comercial "Pense diferente"da Apple, 1997)

BATISTA, Raphael Negrisoli Batista. **Arquitetura de Balanceamento de carga em ambientes paralelos distribuídos utilizando sistemas multiagentes.** 2011. 80 f. Trabalho de Curso (Bacharelado em Ciência da Computação) - Centro Universitário Eurípides de Marília, Fundaçao de Ensino "Eurípides Soares da Rocha", Marília, 2011.

RESUMO

Um ambiente paralelo distribuído é caracterizado por um conjunto de computadores autônomos interconectados por uma rede de computadores de alta velocidade. Este ambiente é capaz de promover bom desempenho em relação ao baixo custo de aquisição, além de boa flexibilidade e manutenção. Porém, o bom desempenho destes ambientes está altamente ligado a eficácia do escalonamento de processos, este que por sua vez, deve distribuir os processos entre os *hosts*, o que não se torna trivial em um ambiente heterogêneo sobrecarregando alguns *hosts* e deixando outros ociosos. O balanceamento de cargas pode disseminar os processos de forma proporcional a capacidade computacional de cada *host*, reduzindo os impactos causados pela heterogeneidade, para isto foi utilizado um sistema multiagente para monitoramento e ativação do balanceamento de cargas. Este trabalho propõe o estudo e desenvolvimento de uma arquitetura de balanceamento de cargas para aplicações paralelas distribuídas que utiliza sistemas multiagentes como suporte ao processo de balanceamento.

Palavras-chave: Sistemas Distribuídos, Balanceamento de Carga, Sistemas Multiagentes.

BATISTA, Raphael Negrisoli Batista. **Arquitetura de Balanceamento de carga em ambientes paralelos distribuídos utilizando sistemas multiagentes.** 2011. 80 f. Trabalho de Curso (Bacharelado em Ciência da Computação) - Centro Universitário Eurípides de Marília, Fundaçao de Ensino "Eurípides Soares da Rocha", Marília, 2011.

ABSTRACT

A distributed parallel environment is characterized by a set of autonomous computers interconnected by a network of high speed computers. This environment is capable of promoting good performance in relation to low cost, and good flexibility and maintenance. However, the good performance of these environments is highly dependent on the scheduling of processes, and that in turn, must distribute the processes among the *hosts*, which is not become trivial in a heterogeneous environment, overloading some *hosts* and leaving others stranded. The load balancing process can spread the proportion computing capacity of each *host*, reducing the impacts caused by heterogeneity. For this was used a multiagent system to monitoring and activate the load balancing. This paper proposes the architecture and development of a multiagent load balancing for distributed parallel applications that uses multiagents systems to support the balancing process.

Keywords: Distributed Systems, Load Balancing, Multiagent Systems.

Lista de Figuras

Sumário

Introdução

O surgimento da computação paralela distribuída deu-se pela crescente demanda de poder computacional para a resolução de questões, sobretudo científicas, que exigem alto consumo de processamento em um tempo menor do que o realizado por computadores sequenciais, a um baixo custo em relação à aquisição de supercomputadores. Um sistema distribuído é um conjunto de computadores independentes que apresenta a seus usuários como um sistema único e coerente (TANENBAUM; STEEN, 2007).

Os sistemas distribuídos interligam vários computadores autônomos, também chamados de nós ou *hosts*, conectados por uma rede de comunicação, de maneira que uma tarefa com alto consumo de CPU seja dividida em vários processos onde cada processo é executado em um nó.

Embora as vantagens obtidas com a computação paralela distribuída sejam evidentes, diversos novos problemas emergem dessa nova abordagem (BRANCO, 2004).

Como afirma Branco (2004) o escalonamento de processos influencia diretamente no desempenho do sistema. Uma vez que o mesmo é responsável por alocar as tarefas entre os processadores. Se esta carga não for escalonada levando em conta o poder computacional de cada *host* em um ambiente heterogêneo, na maioria dos casos, alguns *hosts* podem finalizar o processamento antes de outros de menor capacidade, deixando alguns processadores sobrecarregados, enquanto outros estão quase ociosos.

O escalonador de processos faz a distribuição de processos (tarefas) entre os *hosts* e utiliza o balanceamento de carga para diminuir o efeito das diferenças de velocidade e da capacidade dos *hosts* heterogêneos (MARQUES, 2008). O resultado desta administração e redistribuição é uma possível melhora do desempenho e da eficiência de execução da aplicação (SHIVARATRI; KRUEGER; SINGHAL, 1992).

Para auxiliar as decisões do serviço de balanceamento, técnicas de inteligência artificial (IA) podem ser empregadas. Uma das subáreas da IA denominada Inteligência Artificial Distribuída (IAD) compreendendo o estudo de modelos e técnicas para a resolução da classe de problemas cuja a distribuição seja física ou funcional. Uma técnica favorável neste processo são os Sistemas Multiagentes (SMA), as quais empregam a metáfora de inteligência baseada no comportamento social humano, de forma que um conjunto de entidades inteligentes, denomi-

nados agentes, possam executar a tomada de decisões de modo coordenado em uma sociedade, afim fim de se obter um comportamento global coerente (REZENDE, 2003).

Dessa forma, o presente projeto de pesquisa propõe o estudo e o desenvolvimento de uma arquitetura que aplique o balanceamento de carga dando o suporte para aplicações paralelas distribuídas utilizando SMA no monitoramento dos agentes e na classificação de índices de carga e na otimização da distribuição da carga dentro desta arquitetura.

O objetivo deste trabalho é desenvolver uma aplicação distribuída provida de um escalonador de processos que realize o balanceamento de carga garantindo o equilíbrio do sistema. Para obter o balanceamento, o sistema contará com o auxílio de um sistema multiagentes que monitorará e classificará os *hosts* com índices de desempenho, através do qual serão realizadas trocas de cargas entre os *hosts*, realocando tarefas de nós sobrecarregados em nós ociosos.

Esta monografia está organizada da seguinte forma:

Capítulo 1 – Sistemas computacionais distribuídos – Abordagem dos conceitos essenciais da computação paralela distribuída, como as principais arquiteturas, a comunicação, segurança e o escalonamento de processos com enfase no balanceamento de carga.

Capítulo 2 - Escalonamento de processos e balanceamento de cargas - Este capítulo aborda os principais pontos do escalonamento de processos, políticas de escalonamento e alguns mecanismos para o mesmo. Também descreve o processo de balanceamento de cargas.

Capítulo 3 – Inteligência Artificial Distribuída - Abordagem sobre agentes inteligentes, classificando-os e descrevendo suas principais características. Também apresenta as duas subareas da IAD a Resolução distribuída de Problemas e os Sistemas Multiagenstes.

Capítulo 4 - Desenvolvimento e Resultados - Apresentado as tecnologias utilizadas na implementação do projeto, descrevendo a metodologia utilizada, descrevendo o algoritmo desenvolvido, os resultados encontrados e as conclusões do projeto.

1 Sistemas Computacionais Distribuídos

A história dos sistemas distribuídos teve sua origem no desenvolvimento de computadores multiusuários e das redes de computadores nos anos 60 e foi estimulada pelo desenvolvimento dos computadores pessoais de baixo custo, redes locais (LANS) e o sistema operacionais UNIX nos anos 70 (COULOURIS; DOLLIMORE; KINDBERG, 2007).

Existem várias definições na literatura sobre sistemas distribuídos, entre elas destacam-se a de Tanenbaum e Steen (2007), que definem sistemas distribuídos como um conjunto de computadores independentes que apresenta a seus usuários como um sistema único e coerente e também a de Coulouris, Dollimore e Kindberg (2007), que definem um sistema distribuído como componentes de hardware e software, localizados em computadores em rede se comunicam e coordenam suas ações por meio da troca de mensagens.

Portanto, um sistema distribuído (SD) pode ser definido com um conjunto de computadores autônomos interligados por uma rede de comunicação local ou global, agrupando a capacidade de processamento de cada *host* para realizar determinada tarefa de forma transparente e coerente ao usuário do sistema, aparentando que a tarefa é executada por um único computador.

1.1 Características dos sistemas distribuídos

No desenvolvimento de pesquisas científicas e aplicações comerciais específicas, muitas vezes é necessário grande poder computacional e os sistemas distribuídos mostram-se uma alternativa eficaz e de baixo custo, em comparação aos sistemas centralizados. Um sistema distribuído pode oferecer várias vantagens em relação a um sistema centralizado, Tanenbaum e Steen (2007) cita algumas das principais:

- Economia: o custo de interligar vários computadores se torna mais barato do que a aquisição de um supercomputador, além de poder aproveitar grande parte do processamento

de computadores potencialmente ociosos;

- Velocidade: um sistema distribuído pode apresentar poder computacional igual ou superior a de um supercomputador centralizado;
- Distribuição inerente: os computadores podem estar distribuídos em espaços geográficos diferentes, conectados por uma rede de comunicação global;
- Confiabilidade: caso algum dos computadores apresente alguma falha, o sistema como um todo ainda pode se recuperar e continuar executando a aplicação;
- Crescimento incremental: o poder de processamento pode ser ampliado adicionando novos equipamentos.

Apesar de um SD ter várias vantagens sobre os sistemas centralizados, deve-se analisar alguns problemas que não existiam nos sistemas centralizados. São eles (TANENBAUM; STEEN, 2007):

- Software: o desenvolvimento de softwares é mais complexo, e existem poucos sistemas para SD no mercado;
- Segurança: dificuldade em controlar o acesso de uma grande quantidade de usuários ao sistema;
- Rede de comunicação: a rede pode provocar falhas de comunicação na aplicação, e pode sofrer sobrecarga, não dando vazão à demanda.

Algumas características essenciais em SDs são citadas por Tanenbaum e Steen (2007), Coulouris, Dollimore e Kindberg (2007), Mullender (1993):

- Transparência;
- Flexibilidade;
- Segurança;
- Desempenho;
- Escalabilidade;

1.1.1 Transparência

Uma das características mais importantes de um sistema distribuído é ocultar o fato que seus processos e recursos estão fisicamente distribuídos em vários computadores, dando a impressão que todo o processo ocorre localmente. Os tipos mais importantes de transparência são:

- Transparência de acesso: permite que recursos locais e remotos sejam acessados da mesma forma.
- Transparência de localização: permite o acesso aos recursos ocultando o local físico onde estão alocados.
- Transparência de migração: permite mover recursos entre os *hosts* sem afetar a execução do sistema ou de uma ação de usuário.
- Transparência de relocação: permite transferir recursos para uma outra localização enquanto em uso.
- Transparência de replicação: permite múltiplas instancias de um recurso para aumentar a disponibilidade ou melhorar o desempenho colocando uma cópia perto do local em que ele é acessado.
- Transparência de concorrência: permite que processos distintos operem concorrentemente usando recursos compartilhados sem que um interfira na execução outro.
- Transparência de falha: oculta falhas permitindo que usuários ou aplicações completem suas tarefas apesar da falha de hardware ou software.

1.1.2 Flexibilidade

O software não deve se limitar a um padrão físico de hardware ou de sistema operacional.

1.1.3 Segurança

Um software distribuído deve estar hábil a corrigir possíveis falhas de um computador. Caso um computador obtiver um erro inesperado, ele não pode por em risco toda a aplicação. Além disso, deve manter o controle sobre os usuários que tem acesso ao sistema.

1.1.4 Desempenho

A aplicação distribuída deve possuir um ganho maior de processamento do que softwares de sistemas centralizados. Um dos grandes problemas é o trafego de dados através da rede. Em um sistema centralizado o trafego dos dados ocorre de forma rápida e deve-se garantir que a rede não seja um atraso para a aplicação.

1.1.5 Escalabilidade

A escalabilidade de um sistema pode ser medida no mínimo em três diferentes dimensões. Em primeiro lugar o sistema deve ser escalável em relação ao seu tamanho, podendo adicionar novos usuários e recursos. Em segundo lugar, um sistema escalável em termos geográficos, onde usuários e recursos podem estar geograficamente separados. Em terceiro lugar o sistema deve ser escalável em termos administrativos, sendo fácil de gerenciar mesmo com envolvendo várias organizações. O comprimento de uma ou mais dessas dimensões pode acarretar na perda de desempenho na medida em que o sistema é ampliado.

1.1.6 Middleware

Os sistemas distribuídos podem suportar computadores, redes, linguagens de programação e sistemas operacionais heterogêneos. Essas variedades são mascaradas utilizando protocolos de comunicação e padrões para manipulação de dados. Essa organização costuma acontecer em uma camada de software situada entre a aplicação distribuída e os recursos do sistema operacional. Essa camada é denominada *middleware*.

O termo *middleware* se aplica a uma camada de software que fornece uma abstração de programação, assim como o mascaramento da heterogeneidade das redes, hardware, de sistemas operacionais, e linguagens de programação subjacentes (COULOURIS; DOLLIMORE; KINDBERG, 2007).

A figura 1.1 apresenta a camada de *middleware* se estendendo por várias máquinas oferecendo a mesma interface para cada camada de aplicação, onde a aplicação B é distribuída entre os computadores 2 e 3. Os sistemas distribuídos proporcionam um meio de comunicação entre os componentes de uma aplicação, mas também permite a comunicação entre diferentes aplicações.

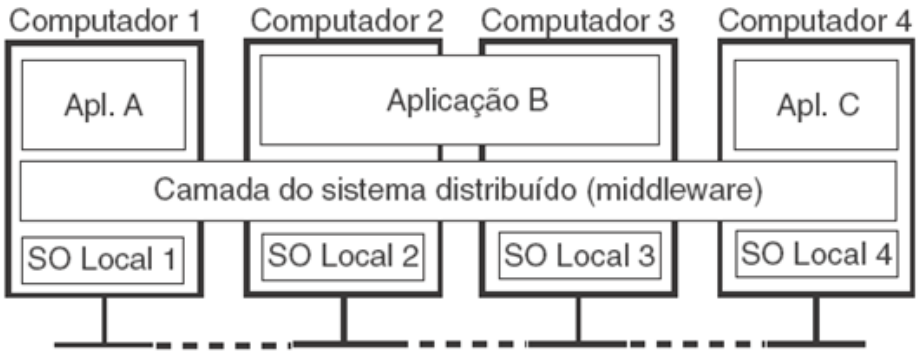


Figura 1.1: Sistema distribuído organizado como *middleware*
(TANENBAUM; STEEN, 2007)

1.2 Arquitetura

Levando em consideração que sistemas distribuídos muitas vezes possuem complexos componentes de software disseminados em várias máquinas é crucial que esses sistemas estejam organizados adequadamente. Uma forma de organizá-los é realizar uma distinção entre os componentes lógicos e físicos destes sistemas (TANENBAUM; STEEN, 2007).

1.2.1 Estilos Arquitetônicos

As arquiteturas de software especificam como vários componentes de software são organizados e como suas interações acontecem. Durante pesquisas de arquitetura de software estabeleceram-se alguns estilos de arquiteturas. Estes estilos arquitetônicos são formulados em termos de componentes, do modo como são conectados, como as trocas dos dados são realizadas e na configuração dos componentes para se tornar um sistema.

Um componente segundo a (OMG, 2004) é uma unidade modular com interfaces requeridas e fornecidas bem definidas que é substituível dentro de seu ambiente. Um termo importante de componentes definido acima, é que ele pode ser substituído desde que suas interfaces sejam respeitadas.

Um conector é descrito como um mecanismo que serve como mediador da comunicação ou da cooperação entre componentes (Mehta et al., 2000; Shaw e Clements, 1997).

Segundo Tanenbaum e Steen (2007), os seguintes estilos arquitetônicos são importantes para sistemas distribuídos:

- Arquitetura em camadas: os componentes organizados em camadas onde um componente

da camada Ni pode realizar requisições somente a camada subjacente N-1, mas não o contrário, como mostra a figura 1.2 (a).

- Arquitetura baseada em objetos: cada componente são organizados em objetos que são conectados por meio de chamadas de procedimento remotos, que são ilustradas na figura 1.2 (b).
- Arquitetura baseadas em eventos: processos se comunicam através da propagação de eventos que opcionalmente também podem transportar dados. Como mostra a figura 1.3 (a).
- Arquiteturas centradas em dados: nesta arquitetura os processos se comunicam por meio de um repositório comum. Este estilo é mostrado na figura 1.3 (b)

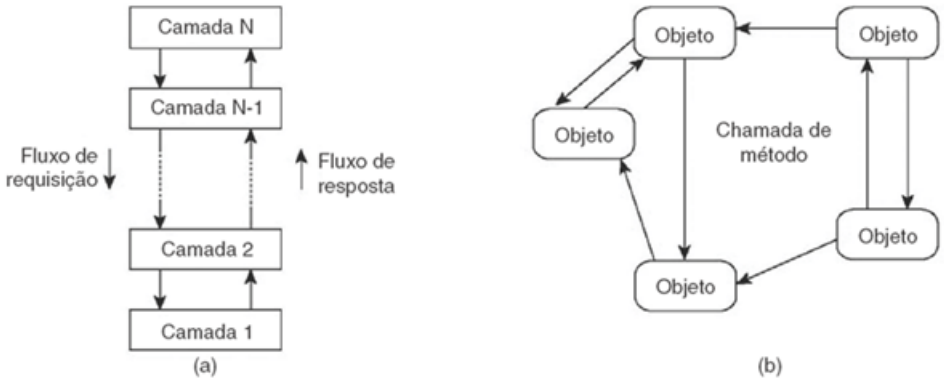


Figura 1.2: Estilo arquitetônico (a) em camadas e (b) baseado em objetos.
(TANENBAUM; STEEN, 2007)

Figura 1.3: Estilo arquitetônico (a) baseado em eventos e (b) de espaço de dados compartilhados.
(TANENBAUM; STEEN, 2007)

Essas arquiteturas de software visam obter um nível razoável de transparência na distribuição e resolver problemas na comunicação e sincronização ente os componentes.

1.2.2 Arquiteturas de sistema

A divisão de responsabilidades entre componentes de um sistema e a atribuição destes nos diversos computadores de uma rede talvez seja o aspecto mais evidente do processo de sistemas distribuídos. Isso tem implicações importantes no desempenho, na confiabilidade e na segurança do sistema resultante (COULOURIS; DOLLIMORE; KINDBERG, 2007).

A concretização de um sistema distribuído consiste na especificação e implantação dos componentes de software em máquinas reais. Existem várias formas de realizar esta tarefa. A especificação final de uma arquitetura de software também é denominada arquitetura de sistemas. Os dois principais tipos de arquiteturas de sistemas são as arquiteturas centralizadas e as descentralizadas.

Arquiteturas centralizadas

Neste modelo os processos do sistema distribuído são divididos em dois grupos Clientes e Servidores. Um servidor é um processo que implementa um serviço específico e um cliente é um processo que realiza uma requisição de um serviço a um servidor. A figura 1.4 mostra o comportamento desta arquitetura.

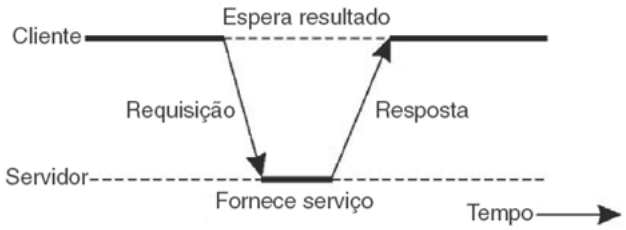


Figura 1.4: Integração geral entre um cliente e um servidor.
(TANENBAUM; STEEN, 2007)

Arquiteturas descentralizadas

Nessa arquitetura, todos os processos envolvidos em uma tarefa ou atividade desempenham funções semelhantes, interagindo cooperativamente como pares (*peers*), sem distinção entre processos clientes e servidores (COULOURIS; DOLLIMORE; KINDBERG, 2007). O objetivo das arquiteturas *pear-to-pear* é aproveitar ao máximo os recursos de vários computadores para o comprimento de uma tarefa. Este tipo de arquitetura é mais escalável do que a arquitetura cliente-servidor, em termos de compartilhamento de serviços, centralizando estes recursos em

um único computador que contém largura de banda e conexões limitadas.

A figura 1.5 ilustra a organização de uma aplicação baseada na arquitetura *pear-to-pear*. São compostas de vários processos executados em computadores diferentes, e a comunicação depende inteiramente da função da aplicação. A figura exibe o compartilhamento entre de vários objetos e dados e um computador individual que contém uma pequena parte da base de dados deste aplicativo.

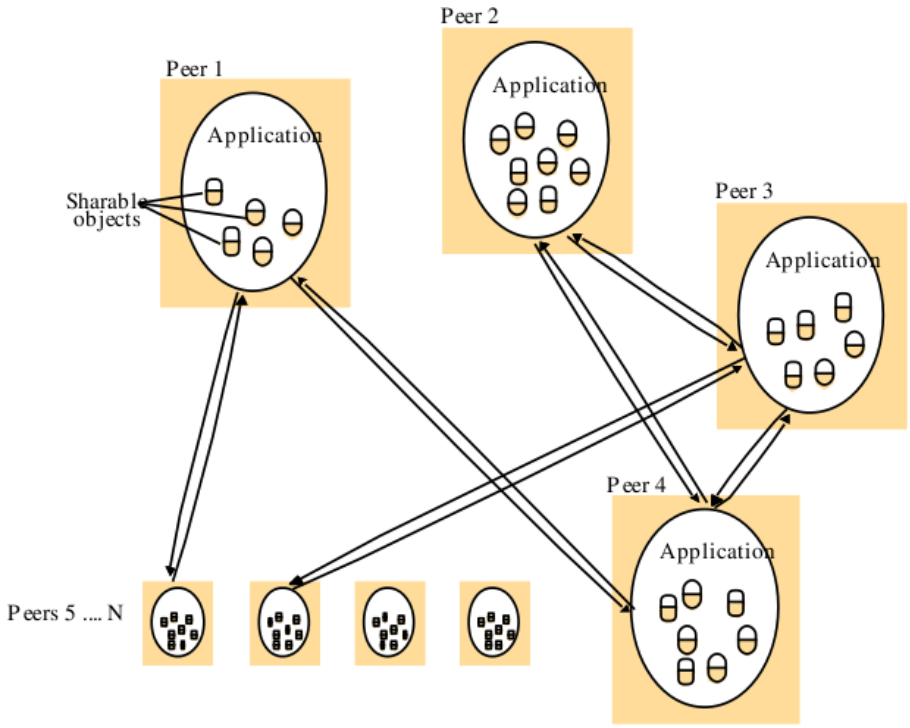


Figura 1.5: Aplicativo distribuído baseado em *pear-to-pear*.
(COULOURIS; DOLLIMORE; KINDBERG, 2007)

1.3 Comunicação

A comunicação entre processos é uma parte fundamental que compõe os SDs, através de redes de comunicação local e/ou global, os processos realizam sua comunicação através da troca de mensagens. A comunicação deve obedecer algumas regras que são denominadas protocolos de comunicação de rede.

> Os sistemas distribuídos utilizam redes locais, redes de longa distância e redes interligadas para comunicação. O desempenho, a confiabilidade, a escalabilidade, a mobilidade e a qualidade das características do serviço das redes subjacentes afetam o comportamento dos sistemas distribuídos e, assim, têm impacto sobre seu projeto (COULOURIS; DOLLIMORE; KINDBERG, 2007).

A troca de mensagens entre dois processos está fundamentada em duas operações *send* e *receive*. Na comunicação entre dois processos, o emissor envia (*send*) uma mensagem (segmento de bytes) ao processo de destino, que recebe (*receive*) a mensagem. Cada processo é associado uma fila onde as mensagens são armazenadas. Os processos remetentes enviam uma mensagem a uma fila remota associada ao processo de destino, este que por sua vez consome a mensagem em sua fila local. Esta operação de troca de mensagens pode ocorrer de duas formas, síncrona ou assíncrona.

Nas trocas de mensagens síncronas os processos participantes na comunicação são sincronizados a cada troca de mensagens. Ao enviar (*send*) a mensagem, o processo emissor é bloqueado até que a mensagem seja recebida (*receive*) no processo de destino.

No caso de mensagens assíncronas, os processos envolvidos na comunicação continuam sua execução normal após a emissão (*send*) e a recepção (*receive*) das mensagens.

As informações armazenadas nos programas em execução são representadas como estruturas de dados - por exemplo, pela associação de um conjunto de objetos -, enquanto que as informações contidas nas mensagens são sequencias puras de bytes (COULOURIS; DOLLIMORE; KINDBERG, 2007).

Ou seja, para que seja possível a transmissão de dados de programas de alto nível através da troca de mensagens, é necessário que haja um simplificação dos dados, convertendo os objetos para uma sequencia de *bytes* antes do início da transmissão dos dados e a reconstrução dos objetos na chegada da mensagem.

Para que seja possível transmitir dados entre processos situados em computadores heterogêneos estes devem realizar a comunicação em um formato comum aos processos. Um padrão aceito para a representação de estruturas de dados e valores primitivos é chamado de representação externa de dados.

Uma forma conveniente de transformar itens de dados e dados primitivos para representação externa dos dados é através do empacotamento dos dados. O processo inverso é o desempacotamento dos dados que consiste em transformar a representação externa dos dados em valores primitivos e em estrutura de dados.

1.3.1 Comunicação cliente-servidor

Nos casos mais comuns entre as iterações entre cliente-servidor, a comunicação de requisições e respostas ocorre de forma síncrona, onde o cliente é bloqueado até que a resposta do

servidor seja recebida. A resposta da requisição é uma confirmação de que a requisição chegou ao servidor, sendo assim é uma forma confiável de comunicação.

A comunicação cliente-servidor trabalha com o protocolo requisição-resposta que é baseado em três primitivas de comunicação: *doOperation,* getRequest *e* sendReply como mostra a figura 1.6.

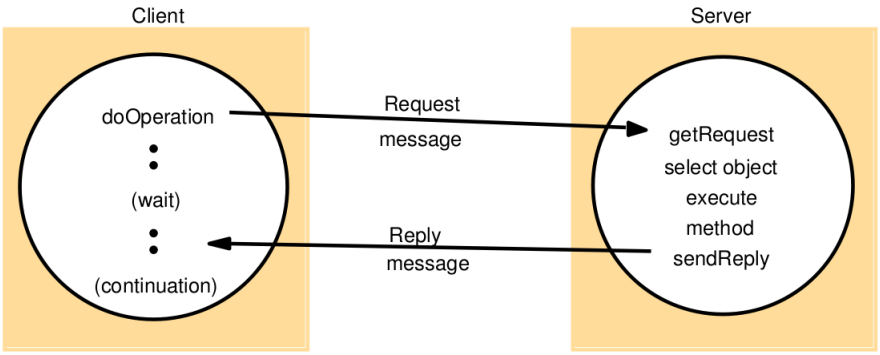


Figura 1.6: Comunicação requisição-resposta.
(COULOURIS; DOLLIMORE; KINDBERG, 2007)

O método *doOperation* é utilizado pelos clientes para invocar operações remotas. Nessa invocação deve ser fornecido o objeto remoto e o método a ser executado juntamente com os parâmetros necessários para a execução do método. O método *doOperation* realiza o empacotamento dos dados e envia uma mensagem de requisição para o servidor. Após o envio da mensagem o método realiza um *receive* para obter uma mensagem de resposta, a partir da qual obtém os resultados e o retorna para o processo que iniciou a requisição. O processo que iniciou a requisição é bloqueado até que o objeto remoto no servidor execute a operação solicitada e transmita uma mensagem de resposta.

O *getRequest* é usado pelo processo servidor para obter as requisições realizadas pelos processos clientes. Após o servidor realizar a operação solicitada na requisição, ele irá utilizar o *sendReply* para enviar a mensagem de resposta para o cliente. Ao receber a mensagem, o cliente será desbloqueado e continuará a execução do programa.

1.3.2 Comunicação em grupo

Um tópico importante da comunicação em SDs é o suporte para o envio de mensagens a vários receptores, também conhecido como comunicação *multicast*.

Uma questão importante é o estabelecimento de caminhos de comunicação para a disseminação das mensagens. Estabelecer estes caminhos não era uma tarefa trivial, que se tornou mais

simples com o avanço das redes *pear-to-pear* e principalmente o gerenciamento estruturado de sobreposição.

A ideia básica do *multicasting* é que os *hosts* se organizem em uma rede se sobreposição, ou seja, uma rede na qual os nós são formados pelos processos e os enlaces representam possíveis comunicações, a princípio, um processo não pode se comunicar diretamente com outro processo, mas deve enviar mensagens através dos canais de comunicação disponíveis. Esta rede de organização é utilizada para disseminar as mensagens para os seus membros.

Tanenbaum e Steen (2007) definem que existem duas formas de organizar a rede de sobreposição, na primeira delas, os nós se organizam em forma de árvore, o que significa que existe somente um caminho entre cada par de nós. Uma outra alternativa é que os nós se organizem em forma de malha, na qual cada nó terá vários vizinhos, e em geral existem vários caminhos entre cada par de nós. A principal diferença entre elas, consiste que a última oferece mais robustez: se algum nó falhar interrompendo a conexão, ainda assim haverá caminhos para propagar as mensagens sem ter que reorganizar toda a rede de sobreposição.

Dessa forma, uma mensagem que é enviada (*send*) a raiz da árvore será propagada (*receive*) a todos os nós subjacentes e assim por diante até os nós folha da árvore.

1.4 Segurança

Existe a necessidade de garantir a privacidade, a integridade e a disponibilidade de recursos em sistemas distribuídos. Coulouris, Dollimore e Kindberg (2007) afirmam que a necessidade de mecanismos de segurança em sistemas distribuídos surge do desejo de compartilhar recursos. A criptografia fornece a base para a autenticação de mensagens, assim como sua privacidade e integridade, dessa forma pode-se garantir que somente agentes autorizados podem enviar e receber mensagens e acessar recursos do sistema. Para isto é necessário o uso de protocolos e políticas de segurança. A escolha do algorítimo de criptografia e do sistema gerenciador de chaves deve ser feita com cuidado pois influencia diretamente no desempenho e na eficácia dos mecanismos de segurança.

A figura 1.7 ilustra o gerenciamento das políticas de acesso de usuários a um objetos. Os usuários podem executar programas clientes que enviam ao servidor para realizar operações sobre objetos. O servidor executa a operação requisitada em cada invocação e envia o resultado para o cliente. Como os objetos podem conter informações confidenciais, o servidor permite o acesso de programas clientes a objetos conforme os *direitos de acesso*, que define quem pode executar determinada ação sobre um objeto. Dessa forma os usuários devem ser incluídos

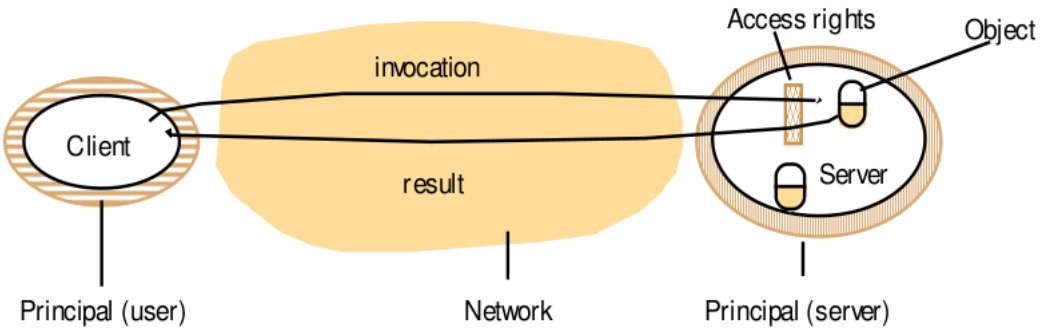


Figura 1.7: Política de acesso a objetos e principais
(COULOURIS; DOLLIMORE; KINDBERG, 2007)

no modelo de segurança como beneficiários dos direitos de acesso. A associação do tipo da autorização e quem a executa acontece a cada requisição e a cada resposta, tal autorização é chamada de *principal*. O servidor deve verificar a identidade do principal do cliente que está realizando a requisição e conferir se ele tem permissões de acesso ao recurso solicitado. O cliente por sua vez pode verificar a *principal* do servidor de modo a garantir que o resultado seja realmente enviado de um servidor pertencente ao sistema.

Os processos interagem enviando mensagens. Estas mensagens são transmitidas através da rede e dos serviços de comunicação, assim ficam expostas a ataques, pois o acesso a rede e ao serviço de comunicação é livre para permitir que dois processos interajam. Usuários mal-intencionados podem realizar ataques externos visando informações ou comprometer a integridade do sistema, especialmente em sistemas que manipulam informações financeiras, confidenciais ou secretas. Um invasor pode enviar mensagens para qualquer processo, ler ou copiar qualquer mensagem enviada entre dois processos, como pode ser visto na figura 1.8. Estes ataques podem ser provenientes de simples computadores ligados a rede munido de um programa que intercepte mensagens enviadas na rede ou que gere mensagens que façam falsas requisições para serviços maquiando ser usuários autorizados.

Como um servidor recebe requisições de vários clientes diferentes, não é simples a tarefa de identificar se as requisição foram realizadas por clientes autorizados ou por um invasor, mesmo que ele exija a inclusão da identidade do principal em cada evocação, visto que um invasor pode gerar uma principal com uma identidade falsa.

Um cliente não consegue identificar se as informações recebida foram retornadas de um servidor legítimo, pois um invasor pode estar fazendo um *spoofing* do servidor. O *spoogfing* é o roubo da identidade de um *host*, assim o cliente pode receber uma resposta que não tenha vindo

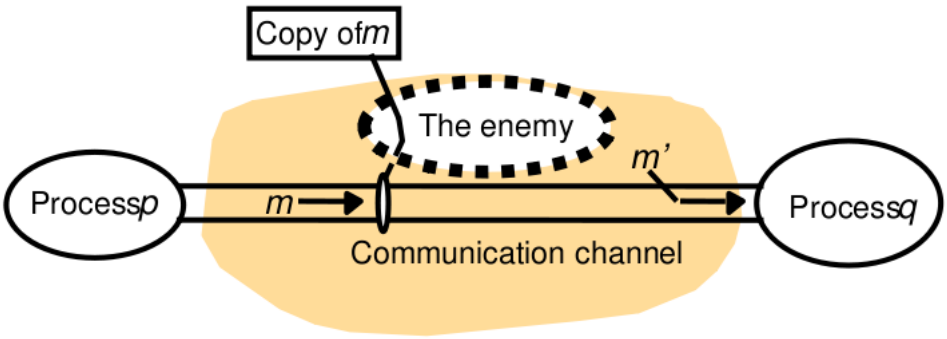


Figura 1.8: Invasor
(COULOURIS; DOLLIMORE; KINDBERG, 2007)

realmente do servidor.

1.4.1 Chaves Compartilhadas

Uma forma de verificar a veracidade das mensagens é compartilhando uma chave conhecida somente entre dois processos. Assim, uma mensagem trocada entre os dois processos que inclui informações dessa chave compartilhada por parte do remetente, dará ao destinatário a certeza de que o remetente é o outro processo do par.

Este segredo compartilhado entre os pares deve receber cuidados para garantir que não seja revelado a nenhum invasor. *Criptografia* é a ciência de manter as mensagens seguras e *cifrar* é o processo de embaralhar uma mensagem de maneira a ocultar seu conteúdo (COULOURIS; DOLLIMORE; KINDBERG, 2007).

1.4.2 Autenticação

Com o uso de chaves compartilhadas e da criptografia o sistema possui uma base para a autenticação de mensagens, ou seja, é possível reconhecer a identidade do remetente das mensagens. Para realizar a autenticação inclui-se na mensagem uma parte cifrada que possua conteúdo suficiente para garantir a sua autenticidade. A autenticação de requisições a servidores poderia, por exemplo, possuir uma representação da identidade do *principal* do *host* que está solicitando o acesso ao objeto, a identificação do objeto e a data e a hora do pedido, tudo cifrado com uma chave secreta compartilhada entre o servidor que está fornecendo este recurso e o processo solicitante. O servidor utilizaria a chave para decifrar a requisição e verificaria se os dados fornecidos conferem com a requisição.

1.4.3 Canais seguros

A criptografia e a autenticação são utilizadas para a criação de canais seguros. Um canal seguro é uma conexão que interliga dois processos, cada um dos quais, atua sobre o nome de um principal. Como mostra a figura 1.9.

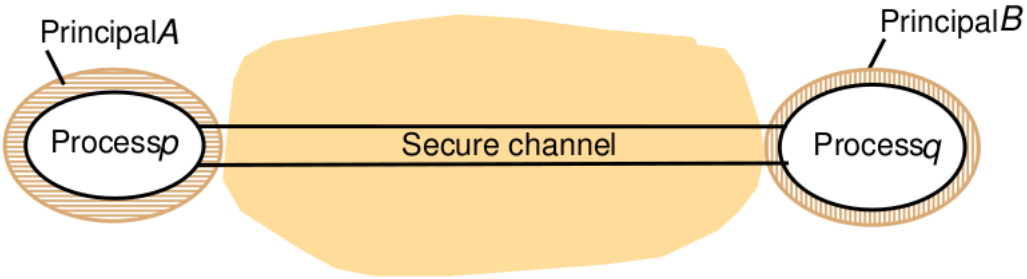


Figura 1.9: Canais seguros
(COULOURIS; DOLLIMORE; KINDBERG, 2007)

Segundo Coulouris, Dollimore e Kindberg (2007), canais seguros possuem as seguintes características:

- Cada processos conhece a identidade do principal do do outro processo a quem está executando.
- Um canal seguro garante proteção contra a falsificação dos dados por ele transmitido.
- Todas as mensagens possuem uma indicação do relógio logico ou físico, para impedir que mensagens sejam reproduzidas ou reordenadas.

O uso das técnicas de segurança como criptografia possui custos de processamento e gerenciamento, dessa forma deve ser analisado o nível de proteção necessário para o sistema.

2 Escalonamento de processos e balanceamento de Cargas

Segundo Tanenbaum (2009) em computadores multiprogramados frequentemente processos competem pela CPU (*Center Process Unit*) ao mesmo tempo. Quando mais de um processo precisa ser executado no mesmo processador, o sistema operacional deve decidir qual deles deve ser executado primeiro. Esta parte do sistema operacional é denominado escalonador de processos.

Em um sistema distribuído a função do escalonador de processos é determinar a ordem de execução dos processos e alocá-los aos *hosts* do sistema. O escalonador de processos realiza a tomada de decisões com base em uma política de escolha, utilizando algoritmos de escalonamento e implementando políticas que determinam, onde e como os processos devem ser alocados.

2.1 Política de escalonamento

As políticas de escalonamento definem critérios e regras para a ordenação das tarefas a serem realizadas para que ocorra o escalonamento. Dividem-se da seguinte forma: política de transferência, política de seleção, política de localização, e política de informação (SHIVARATRI; KRUEGER; SINGHAL, 1992).

A política de transferência identifica se um *host* está em um estado adequado para participar de uma migração, como transmissor ou como receptor.

A política de seleção determina quais os processos localizados em *hosts* sobrecarregados devem ser migrados.

A política de localização determina quais *hosts* ociosos que deverão receber as migrações.

A política de informação é responsável por decidir em que momento as informações a respeito do estado dos *hosts* devem ser coletadas, de onde serão coletadas, e quais informações

serão coletadas (utilizadas). Existem três tipos de políticas de informação (MARQUES, 2008):

- política orientada à demanda: onde uma máquina coleta o estado das demais somente quando ela se torna emissora ou receptora;
- política periódica: as informações são coletadas de tempos em tempos;
- política orientada à mudança de estado: as informações das máquinas são coletadas de acordo com a mudança de seu estado.

2.2 Mecanismos de escalonamento

Os mecanismos respondem pela definição de como o escalonamento deverá ser efetuado. Dividem-se em três categorias: mecanismo de métrica da carga, mecanismo de comunicação da carga e o mecanismo de migração. O mecanismo de métrica da carga define o método utilizado para medir a carga de cada elemento de processamento. O mecanismo de comunicação da carga determina o método por meio do qual será efetuada a comunicação das informações de cargas entre as máquinas disponíveis. E o mecanismo de migração delibera o protocolo que deverá ser utilizado quando ocorrer migração de processos entre máquinas (SHIVARATRI; KRUEGER; SINGHAL, 1992)(BRANCO, 2004).

2.3 Balanceamento de Carga

A melhoria de desempenho é uma das questões mais importantes em sistemas computacionais distribuídos. Uma forma óbvia de se alcançar este resultado, porém de alto custo, é aumentar a capacidade dos nós e adicionar mais nós ao sistema. Estas melhorias podem ser necessárias em casos de sobrecarga de todos os nós do sistema. No entanto, em muitas situações, o baixo desempenho é devido a distribuição desigual de carga em todo o sistema. As vezes, a chegada aleatória de processos em tal ambiente pode resultar na sobrecarga de alguns nós enquanto outros nós estão ociosos ou com pouca carga. A figura 2.1 ilustra esta situação.

O balanceamento de carga melhora o desempenho transferindo tarefas de nós muito carregados, onde exista pouca capacidade de processamento, para nós que estão com a carga mais leve, onde as tarefas podem ser beneficiadas da capacidade computacional, que caso contrário não seriam usadas.

Se a carga de trabalho em alguns é geralmente mais pesada do que a de outros, ou se alguns

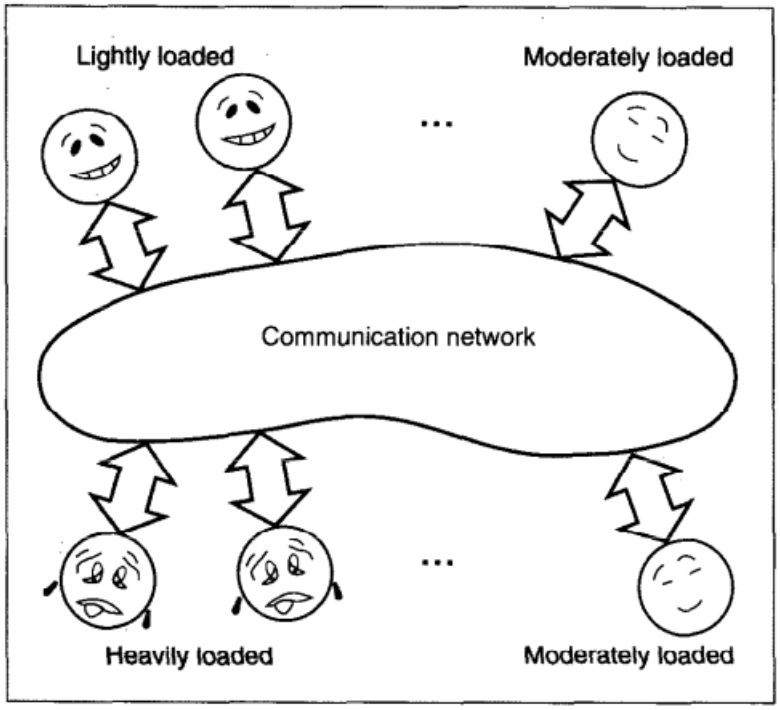


Figura 2.1: Sistema distribuído sem balanceamento de carga
(SHIVARATRI; KRUEGER; SINGHAL, 1992)

nós levam mais tempo para executar tarefas do que outros, situações de desigualdade de cargas entre os nós podem ocorrer frequentemente.

Apresentam-se como fator que pode dificultar a execução de um mesmo código em diferentes máquinas tanto a heterogeneidade arquitetural quanto a heterogeneidade configuracional, pois as características das máquinas são diferentes, fato que pode torná-las incompatíveis, ou até mesmo gerar inconsistências (MARQUES, 2008).

A heterogeneidade deve ser considerada ao efetuar uma alocação de tarefa, uma vez que existem *hosts* com diferentes potências computacionais (BRANCO, 2004). É imprescindível escolher o que melhor se aplica às restrições de determinada tarefa. Para garantir o melhor aproveitamento de ambientes heterogêneos o balanceamento da carga deve ser realizado de acordo com a capacidade de cada *host* do sistema, de forma que todos os *hosts* mantenham uma carga equilibrada, sem sobrecargas e sem ociosidade.

Para isso, o escalonador de processos faz a distribuição de processos (tarefas) entre os *hosts* e utiliza o balanceamento de carga para diminuir o efeito das diferenças de velocidade e da capacidade dos *hosts* heterogêneos. O resultado desta administração e redistribuição é uma possível melhora do desempenho e da eficiência de execução da aplicação (SHIVARATRI; KRUEGER; SINGHAL, 1992).

3 Inteligência Artificial Distribuída

A Inteligência Artificial Distribuída (IAD) emergiu da integração entre as áreas de Inteligência Artificial (IA) e Sistemas Distribuídos (SD) compreendendo o estudo de modelos e técnicas para resolver a classe de problemas cuja a distribuição, física ou funcional, seja inerente.

> Uma característica de tais sistemas é a metáfora de inteligencia baseada no comportamento social, diferentemente da IA que baseia-se no comportamento individual, estabelecendo modelos, arquiteturas e implementações para que um conjunto de entidades inteligentes, denominadas agentes, possam executar ações de modo coordenado em uma sociedade para que no final se obtenha um comportamento global coerente (REZENDE, 2003).

A IAD analisa aspectos sociais e políticos de sistemas computacionais apresentando formas mais abrangentes de resolução de problemas complexos, planejamento, representação de conhecimento, coordenação, comunicação, negociação, etc., permitindo a resolução de problemas de forma cooperativa e distribuída, utilizando processos denominados agentes.

Bond e Gasser (1988) definem IAD como "o campo de IA que tem interesse no uso da concorrência em suas computações"e propõem a divisão de IAD em duas áreas principais: Resolução Distribuída de Problemas que se preocupa com a solução de problemas específicos usando diversos módulos cooperando entre si, e Sistemas Multiagentes, que se preocupa com a coordenação dos comportamentos de diversos agentes inteligentes autônomos para atingir um ou mais objetivos.

O funcionamento da IAD depende de um determinado conjunto de partes (ou módulos) para resolver de modo cooperativo um determinado problema. Sua modularidade para encontrar soluções de problemas está diretamente ligada ao conceito de agentes (BARRETO, 2011).

3.1 Conceitos de Agentes Inteligentes

Existem diversas definições de agentes na literatura, que variam de acordo com a aplicação na qual são empregados. Entre elas destaca-se a de Russell e Norvig (2010) que definem um agente como qualquer coisa que pode perceber seu ambiente através de sensores e agir sobre este ambiente através de atuadores. Para exemplificar esta definição pode-se realizar um comparativo entre robôs, softwares e seres humanos.

Um agente humano possui órgãos como olhos, ouvidos, nariz, entre outros que funcionam como sensores na percepção do ambiente no qual se encontra. E possui braço, pernas, boca, entre outros órgãos que funcionam como atuadores dentro deste ambiente. Um agente robô pode possuir câmeras, sensores infravermelho, GPS, entres outros sensores para identificar o ambiente e pode possuir vários motores como atuadores. Um agente de software percebe seu ambiente através das entradas de teclado, através da leitura de um arquivo, ou pacotes de rede e pode agir sobre seu ambiente exibindo informações através de um monitor ou impressora, gravar informações em arquivo e enviar pacotes através da rede.

Agentes são software que operam sobre ambientes dinâmicos e complexos, os quais atuam autonomamente de acordo com a percepção obtida. Esta ação executada deve realizar um conjunto de tarefas e objetivos do agente (SOUZA, 1996).

> A percepção de um agente é utilizada para representar entradas sensoriais ao agente. A sequência de percepções é um histórico contendo todas as entradas sensoriais de um determinado agente. Esse histórico irá influenciar o agente na tomada de ações. A chamada função do agente mapeia qualquer sequência de percepções para que uma determinada ação seja tomada, definindo um comportamento (RUSSELL; NORVIG, 2010).

Russell e Norvig (2010) afirmam que o comportamento de um agente é representado através de uma função matemática que mapeia qualquer sequência específica de percepções relacionando ações que devem ser executadas pelo agente quando estes pré-requisitos sensoriais forem satisfeitos. Para um agente de software essa função é representada por um programa de agente que consiste na sua implementação concreta.

3.1.1 Classificação de Agentes

Agentes são geralmente divididos em dois grupos: agentes reativos e agentes cognitivos.

Agentes Reativos

A abordagem reativa foi proposta por Brooks (Brooks, R., 1986) no domínio da robótica diante da falta de resultados satisfatórios de robôs cognitivos em ambientes dinâmicos ou desconhecidos. Sua proposta abordava robôs mais simples, baseados em ações elementares ativadas diretamente em resposta a determinados estímulos, segundo uma prioridade pré-definida.

Estes estímulos são alterações sofridas no ambiente percebidas pelo agente que funciona como um gatilho que dispara uma ação no agente com base na execução de regras, atualizações da base de conhecimento, ou espaço de crenças. Um modelo geral de agente reativo pode ser observado na figura 3.1.

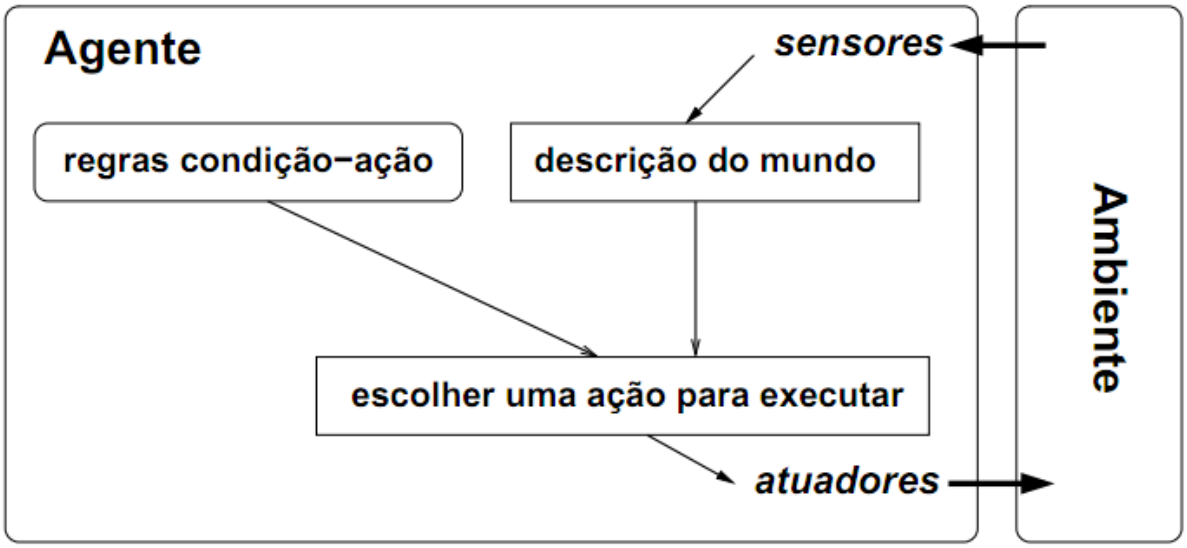


Figura 3.1: Modelo de um agente reativo
(RUSSELL; NORVIG, 2010)

Algumas das principais características dos agentes reativos são (GARCIA; SICHMAN, 2003):

- não há representação explícita de conhecimento: o conhecimento dos agentes é implícito e se manifesta através de seu comportamento;
- não há representação do ambiente: o seu comportamento baseia-se no que é percebido em cada instante do ambiente, mas sem um representação explícita deste;
- não há memória de ações: o resultado de ações anteriores não influencia sobre ações futuras;

- organização etológica: a forma de organização dos agentes reativos é similar a dos animais, ou seja, um agente isoladamente é bem simples entretanto o trabalho realizado por um conjunto de agentes torna-se bem complexo;
- um grande número de membros: geralmente sistemas multiagentes reativos possuem um grande número de agentes, da ordem de dezenas, centenas ou mesmo milhões de agentes.

Agentes Cognitivos

Os agentes cognitivos baseiam-se na organização social humana. Possuem um representação explícita do ambiente e dos demais agentes, armazenam seu histórico de ações permitindo planejar ações futuras. A comunicação entre agentes cognitivos pode ocorrer de forma direta através de linguagem e um protocolo de comunicação complexo, o que é possível devido ao sistema de percepção, que permite examinar o ambiente, e o de comunicação, que permite a troca de mensagens entre agentes, são distintos, o que não ocorre nos agentes reativos. Tipicamente possuem poucos agentes, dado que cada agente é um sistema sofisticado e computacionalmente complexo (HÜBNER; BORDINI; VIEIRA, 2004). A figura 3.2 apresenta um modelo genérico de agente cognitivo.

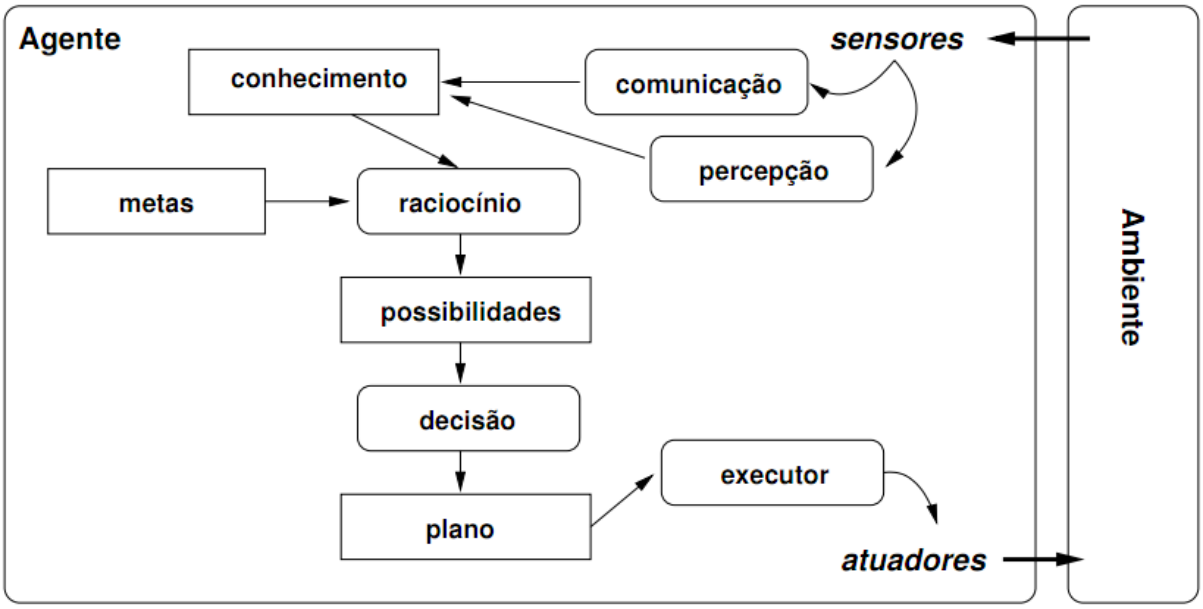


Figura 3.2: Modelo de um agente cognitivo
(DEMAZEAU; MÜLLER, 1990 apud HÜBNER; BORDINI; VIEIRA, 2004)

3.1.2 Racionalidade de Agentes

Um agente racional é aquele que faz a coisa certa - conceitualmente falando, todas as entradas sensoriais definidas na função de agente serão consideradas corretas. É obvio que tomar a decisão certa é melhor do que tomar a decisão errada, mas o que significa fazer a coisa certa? (RUSSELL; NORVIG, 2010)

Um agente racional possui uma ferramenta, denominada Medida de Desempenho (MD), que analisa o nível do sucesso obtido na realização de suas ações. Para isto, após a execução de uma ação a MD verifica o estado resultante do ambiente. A MD se baseará em critérios para medir o nível do sucesso do comportamento do agente.

Segundo Russell e Norvig (2010), existem quatro fatores que determinam a racionalidade de um agente:

- A medida de desempenho que define o grau de sucesso.
- O conhecimento anterior que o agente tem sobre o ambiente.
- Que ações o agente pode realizar.
- O histórico da percepção do agente (sequência de percepções)

Estes fatores compõem a definição de agentes racionais:

Para cada sequência de percepção possível, um agente racional deve saber se sua ação maximizará sua medida de desempenho, baseado na evidência de sua sequência de percepção e no conhecimento que ele traz consigo. (RUSSELL; NORVIG, 2010)

Uma forma de aumentar a medida de desempenho de um agente é realizar a coleta de informações do ambiente onde atua. A realização de ações com a finalidade de modificar percepções futuras é uma parte importante da racionalidade. A coleta de informações também pode ser utilizada como forma de explorar um ambiente desconhecido pelo agente.

3.1.3 Aprendizagem

Um agente racional não apenas coleta informações, mas também aprende tudo quanto está ao alcance de sua percepção. Um agente inicialmente configurado pode possuir um conhecimento prévio sobre o ambiente, mas, como o agente adquire experiência ao longo da coleta de informações o seu conhecimento pode ser modificado e ampliado. Em alguns casos específicos

o agente pode conhecer completamente o ambiente. Neste caso não é necessário percepção nem aprendizado, ele simplesmente age corretamente, porém este tipo de agente se torna muito frágil.

3.1.4 Autonomia

Agentes Autônomos são sistemas computacionais que habitam algum ambiente dinâmico e complexo, percebem e atuam autonomamente neste ambiente e, fazendo isto, atingem um conjunto de objetivos ou tarefas para os quais foram projetados (MAES, 1995).

Quando um agente se baseia somente no conhecimento anterior de seu projetista ele não tem autonomia. Os agentes devem aprender aprender através das experiências adquiridas e alterar seu comportamento de modo a complementar ou corrigir o conhecimento prévio parcial ou incompleto. A autonomia de um agente remete a capacidade de atuar sobre o ambiente por uma motivação própria baseado no nível de sucesso adquirido em suas experiências. A motivação do agente é definida pelo projetista em um atributo denominado propósito.

3.2 Resolução Distribuída de Problemas (Distributed Problem Solving)

A resolução distribuída de problemas (RDP) tem sua motivação inicial na solução de um *problema inicial preciso*, este modelo também é conhecido como quadro-negro (*Blackboard*). A concepção de agentes na RDP é fundamentada na resolução de um problema específico. Durante a resolução deste problema, os agentes são fisicamente distribuídos em diversas máquinas conectadas através de uma rede.

A organização dos agentes, na maioria dos casos, é completamente definida durante a fase de concepção do projeto. Esta organização acontece para restringir o comportamento dos agentes.

A comunicação entre os agentes pode ser realizada através de trocas de mensagens ou pelo compartilhamento de dados comum. A estrutura de comunicação quase sempre é definida durante a fase de concepção do sistema, sendo fortemente ligada ao modelo algorítmico subjacente e ao problema cujo o sistema deverá resolver. Os agentes são executados concorrentemente, de modo a aumentar a velocidade da resolução.

Os agentes cooperam, dividindo em tarefas (ou sub-problemas) as diversas partes do problema original, ou ainda, aplicando diferentes estratégias de resolução na mesma tarefa. Para

garantir um comportamento global coerente existe a noção de um *controle global*, conforme a organização inicial prevista. Este controle pode ser implementado de modo centralizado, deixando um agente responsável por gerenciar o sistema, ou de modo distribuído.

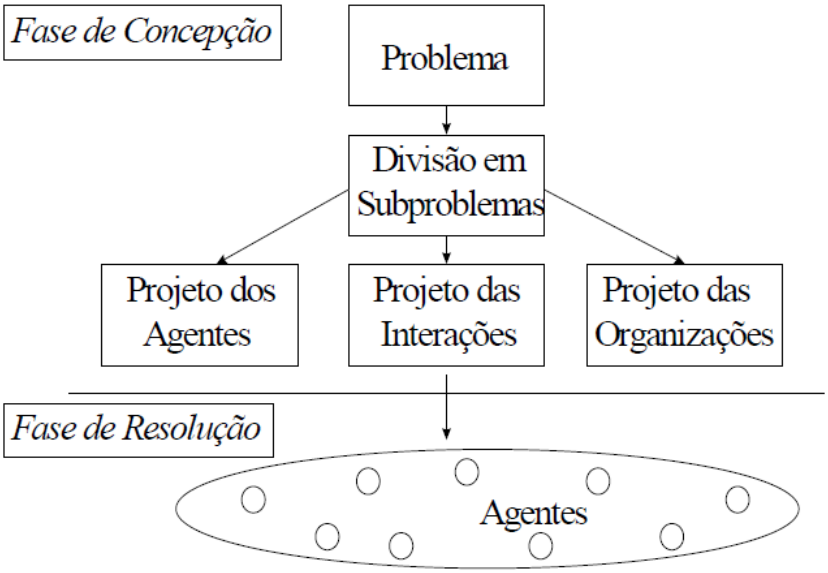


Figura 3.3: A abordagem RDP
(SICHMAN, 1995)

A figura 3.3 representa a abordagem RDP. Do ponto de vista da concepção do projeto, os agentes, a sua organização e sua interação existem somente pela existência de um problema no qual o sistema deva solucionar. Não existindo a preocupação quanto a reutilização dos agentes em outro contexto. RDP aplica técnicas de coordenação e sincronização de sistemas distribuídos a fim de integrar sistemas concebidos em IA. De maneira geral, podemos considerar que os sistemas RDP são o cruzamento destas duas áreas.

3.3 Sistemas Multiagentes

Os sistemas multiagentes (SMA) são compostos de agentes capazes de tomar decisões de forma autônoma que maximizem sua medida de desempenho levando à satisfação do seu objetivo, interagindo com outros agentes presentes no sistema através de protocolos de interação social baseados nos humanos e que possuam algumas características como coordenação, cooperação, competição e negociação (REIS, 2003).

> Não existe um problema definido a priori que o sistema deve resolver, o objetivo da área é estudar modelos genéricos a partir dos quais podem-se conceber agentes, organizações e interações, de modo a poder instanciar tais conceitos

> quando se deseja instanciar tais conceitos quando se deseja resolver um problema em particular. Dito de um outro modo o objetivo é conceber os meios a partir dos quais pode-se assegurar que agentes desejem cooperar e efetivamente o façam, com o intuito de resolver um problema específico quando este for apresentado ao sistema (Alvares, Luis Otavio and Sichman, Jaime Simão, 1997).

Garcia e Sichman (2003) propuseram uma taxonomia na qual um SMA pode ser classificado segundo alguns eixos tais como:

- *Eixo de Abertura*: indica a possibilidade do sistema alterar sua composição dinamicamente, alterando o número de agentes que o compõem;
- *Eixo de Granularidade*: um SMA pode conter poucos agentes (SMA com baixa granularidade), ou milhares de agentes (SMA com alta granularidade);
- *Eixo de Composições*: pode conter agentes homogêneos, agentes idênticos podem tratar diferentes porções do espaço de entrada, ou agentes heterogêneos com tratamento complementares para solucionar um determinado problema.

Um dos pontos essenciais para a construção de uma sociedade de agentes, consiste em coordenar as interações e as dependências de atividades dos diferentes agentes no contexto do Sistema Multiagente. A coordenação desempenha um papel fundamental nos SMAs que são sistemas inerentemente distribuídos. As metodologias de coordenação de agentes é dividido principalmente em duas partes (REIS, 2003):

- Competitivos: agentes preocupados com seu bem próprio, que possuem a coordenação por negociação.
- Cooperativos: agentes que se preocupam com o bem do conjunto. Nestes casos são aplicáveis as metodologias que possibilitam definir uma organização estrutural da sociedade de agentes, a definição de troca de papéis, a definição de alocação de tarefas aos diversos agentes e o planejamento conjunto multiagente.

Os SMA incluem diversos agentes que interagem ou trabalham em conjunto, podendo ser agentes homogêneos ou heterogêneos. Cada agente é basicamente um elemento capaz de resolver problemas de forma autônoma, operando assincronamente com outros agentes. Para garantir a operação dos agentes como parte do sistema é necessário uma infra-estrutura que permita a comunicação entre os agentes que compõem o SMA. Na figura 3.4 pode-se observar a arquitetura de um sistema multiagente.

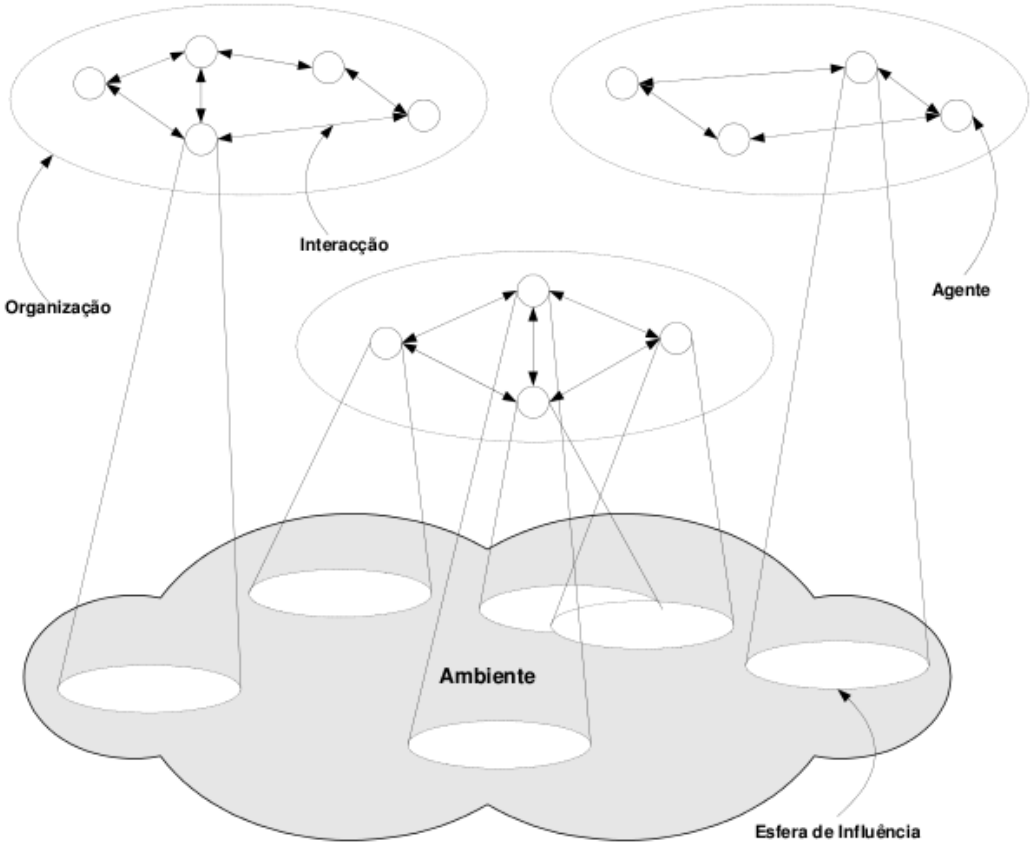


Figura 3.4: Estrutura de um Sistema Multiagente
(REIS, 2003)

Em um SMA existem diversos agentes com capacidades de perceber e agir de modos diferentes no ambiente. Segundo JENNINGS et al. (2000) cada agente possui uma esfera de influência distinta sobre o ambiente, influenciando diferentes partes do ambiente. Conforme as relações existente entre os agentes as esferas de influência podem coincidir. Por exemplo, na geração de horários de dois departamentos, os agentes responsáveis pela geração de cada horário terão como esfera de influência os horários dos docentes que lecionam no seu departamento, das turmas dos cursos desse departamento e das salas alocadas a esse departamento. Esses agentes irão interagir no caso de efetuarem alocações em docentes, turmas ou salas que lhes sejam comuns (REIS, 2003). Estas iterações podem ocorrer de diversas formas.

3.3.1 Comunicação em SMA

A comunicação é parte fundamental para que haja colaboração, negociação e cooperação entre entidades independentes. Em sistemas multiagentes é necessário uma linguagem comum entre todos os agentes presentes no ambiente para compartilhar o conhecimento adquirido e para coordenar as atividades entre os agentes. A linguagem de comunicação define a capacidade de comunicação de cada agente. Deve ser consistente e ter um número limitado de primitivas de comunicação. Segundo Meneses (2001) a comunicação humana e a teoria dos atos comunicativos (*Speech Act Theory*) é tomada como modelo para a comunicação de agentes. Esta teoria usa o conceito de performativas para conduzir suas ações. Um exemplo das linguagem de comunicação de agentes são as linguagens KQML (*Knowledge Query and Manipulation Language*) e FIPA-ACL (*Foundations of Intelligent Physical Agents - Agent Communication Language*).

No contexto dos sistemas multiagentes existem algumas formas de trocar informações (BAKER, 1997).

- Comunicação direta;
- Comunicação assistida;
- Difusão de mensagens;
- Blackboard;

Na comunicação direta agentes trocam mensagens diretamente sem a necessidade de um intermediário. Há o estabelecimento de uma ligaçao direta (ponto-a-ponto) através de um conjunto de protocolos que garantem a entrega da mensagem em segurança (Figura 3.5). Neste tipo de comunicação é necessário que os agentes tenham conhecimento prévio dos agentes presentes no sistema e da forma como endereçar as mensagens. A principal vantagem deste tipo

de comunicação é que o uso de um agente coordenador é prescindível. Agentes coordenadores podem gerar um "gargalo"ou até mesmo o bloqueio do sistema quando há uma grande troca de mensagens. Porém o custo da comunicação é alto principalmente quando há um grande número de agentes no sistema, tornando também a implementação complexa em relação às outras formas de comunicação.

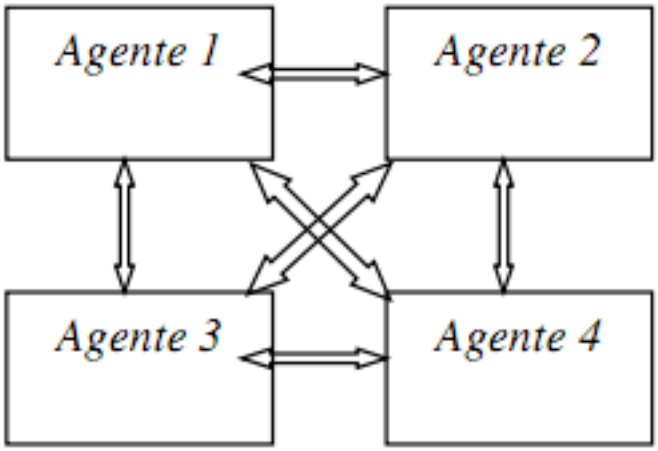


Figura 3.5: Comunicaçao Direta entre agentes
(BAKER, 1997)

Na comunicação assistida ou comunicação por sistemas federados (BAKER, 1997) é definido uma estrutura hierárquica de agentes e a troca de mensagens se dá através de agentes especiais denominados "facilitadores"ou mediadores (Figura 3.6). Esta alternativa reduz o custo de complexidade necessária aos agentes individuais na realizaçao da comunicação. É geralmente utilizado em sistemas com muitos agentes.

Na comunicação por difusão de mensagens ou *broadcast* as mensagens são propagadas para todos os agentes presentes no sistema, geralmente é utilizada em situações onde todos os agentes necessitam receber a informação ou quando o agente remetente não conhece o agente destinatário ou seu endereço.

Segundo Baker (1997) a comunicação por quadro-negro ou *blackboard*, é muito utilizada na área de inteligência artificial como modelo de memória compartilhada. Atua como se fosse um repositório onde agentes escrevem suas mensagens a outros agentes e obtêm informações sobre o ambiente.

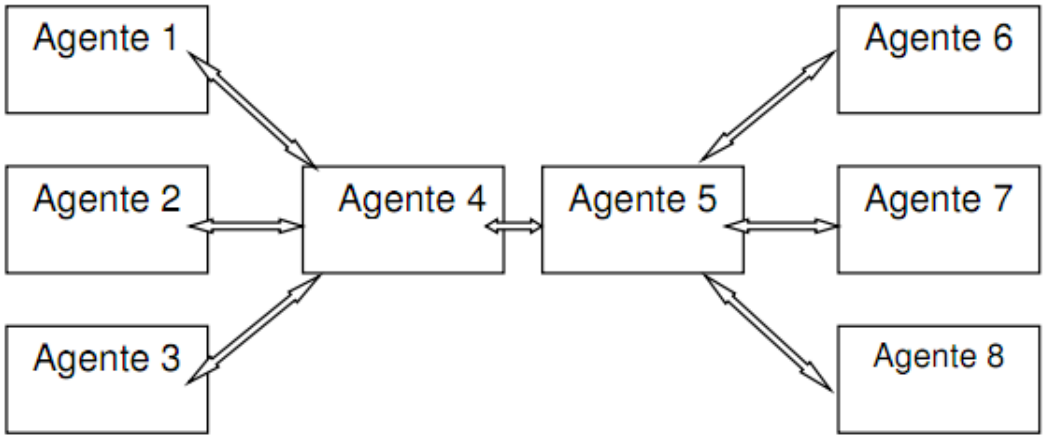


Figura 3.6: Comunicaçao Direta entre agentes
(BAKER, 1997)

3.3.2 Coordenação em SMA

Segundo Weiss (1999) a coordenação é uma característica fundamental para um sistema de agentes que executam alguma atividade em uma ambiente compartilhado.

A coordenação está altamente relacionada com o compartilhamento de conhecimento entre os agentes, com o principal objetivo de coordenar as ações individuais de cada agente para atingir o objetivo final do sistema multiagente. Há também uma preocupação com o comportamento global do sistema, analisando se este está caminhando para a resolução do problema de forma coerente. A coordenação entre agentes dá-se principalmente pelo fato que um agente isolado, dentro de um sistema multiagente, não possuir informação ou capacidade suficiente para resolver grande parte dos problemas, não conseguindo atingir muitos dos objetivos.

A organização de um SMA pode ser vista simplificadamente como um conjunto de restrições adotadas por um grupo de agentes para que possam atingir seus objetivos globais mais facilmente (REZENDE, 2003).

Dessa forma, a coordenação contribui para a sincronização dos objetivos dos agentes de forma a trabalharem em conjunto a fim de concluírem o objetivo final do sistema. Geralmente para uma cooperação bem sucedida, cada agente deve manter um *modelo* dos outros agentes e também desenvolver um modelo de interações futuras ou possíveis. Ela pode ser dividida em cooperação e negociação.

Negociação é a coordenação entre agentes antagônicos ou simplesmente egoísta (*self-interested*) (WEISS, 1999). A negociação é o tipo de coordenação que trabalha com agentes competitivos, ou seja, que possuem objetivos conflitantes. Nestes casos são utilizados protocolos de negocia-

ção para determinar as regras de negociação e são definidos os conjuntos de atributos sobre os quais se pretende chegar a um acordo.

Como afirma Weiss (1999) a cooperação é a coordenação entre agentes não antagônicos. Neste tipo de coordenação os agentes não possuem conflitos em seus objetivos, contribuindo uns com os outros, mesmo que provoquem custos individuais.

A coordenação tem como objetivo principal a sincronização entre objetivos de agentes visando o objetivo final do sistema.

4 Desenvolvimento e Resultados

Neste capítulo serão abordadas as tecnologias utilizadas para o desenvolvimento do projeto e o funcionamento do algoritmo proposto.

4.1 Tecnologias utilizadas

O projeto foi desenvolvido utilizando a linguagem de programação Java na versão 1.7.0 build 147. Para o desenvolvimento dos sistemas multiagentes foi utilizado o *framework* de desenvolvimento JADE (*Java Agent DEvelopment Framework*) na versão 4.1, este *framework* é estruturado de acordo com as especificações da FIPA (*The Foundation of Intelligent Physical Agents*) e auxilia no desenvolvimento de agentes encapsulando comportamentos de agentes, a troca de mensagens que é realizada através do Java RMI (*Remote Method Invocation*) e o serviço de páginas amarelas.

4.1.1 JADE

JADE é um *framework* de desenvolvimento de aplicações baseadas em agentes totalmente implementado na linguagem Java. Ele simplifica a implementação de sistemas multiagentes por meio de um *middleware* implementado conforme as especificações da FIPA e através de um conjunto de ferramentas gráficas que auxilia durante a fase de depuração e desenvolvimento. A plataforma de agentes pode ser distribuída através de máquinas, que não precisam compartilhar do mesmo sistema operacional, e pode ser controlada remotamente através de uma interface visual.

JADE é um software *open source* sob a licença LGPL (*Lesser General Public License Version 2*) e é distribuído por Telecom Itália.

Maiores detalhes podem ser obtidos no site oficial da plataforma: http://jade.tilab.com/

4.2 Especificação do algoritmo

A implementação do algoritmo foi baseado no conceito de sistemas distribuídos do tipo centralizado sendo desenvolvido dois tipos de Agentes: o agente coordenador e o agente cliente. Para implementação dos agentes foi utilizado o conceito de sistemas multiagentes.

O agente coordenador é responsável por gerar e distribuir uma lista de dados a serem processados para cada agente cliente registrado no sistema.

O agente cliente por sua vez recebe os dados a serem processados adicionando-os a uma fila, onde serão consumidos de acordo com a ordem de recebimento. Os agentes clientes agem de forma cooperativa para processarem os dados da forma mais rápida possível a fim de completar o objetivo final do sistema.

Os agentes implementados são do tipo reativo, agindo de acordo com alterações sofridas em seu ambiente tendo seus comportamentos acionados através do recebimento de mensagens para realizar o processamento.

A comunicação entre os agentes ocorre de forma direta, onde o coordenador envia mensagens para cada um dos agentes. Os agentes clientes também realizam notificações de carga diretamente ao coordenador.

Foram estabelecidos como política de transferência o limite mínimo e máximo de carga que o agente cliente poderia manter em sua fila de processamento. Quando um destes limites é atingido o agente cliente envia uma mensagem ao agente coordenador iniciando o processo de balanceamento de cargas.

Ciclicamente um processo é executado em cada cliente verificando a quantidade de tarefas alocadas na fila de processamento e verificando se o cliente está acima do nível de carga ideal, caracterizando uma sobrecarga, ou se está abaixo do nível de tarefas definido, sendo classificado como ocioso. Após esta identificação uma mensagem de notificação é enviada ao agente coordenador.

O serviço de balanceamento de carga é realizado pelo agente coordenador, que ao receber uma mensagem específica de notificação do estado de carga de um cliente o registra em uma fila de acordo com seu nível de carga, após este processo o coordenador verifica se existem agentes nas condições necessárias para realizar a migração dos dados para outro agente.

Ao receber uma mensagem de agente ocioso, o coordenador o registra em uma fila específica. Quando uma mensagem de agente sobrecarregado é recebida o agente coordenador responde a mensagem com o identificador de um agente ocioso, caso exista. Quando o agente

sobrecarregado recebe uma mensagem do coordenador para movimentação de carga, ele envia ao agente indicado pelo o coordenador uma tarefa afim de aliviar sua carga.

Como a política de transferência foi definido que sempre a última tarefa recebida pelo agente sobrecarregado será movida para o novo agente classificado como ocioso.

Estes processos ocorrerão até que todos os dados gerados pelo agente coordenador sejam processados pelos clientes.

4.3 Implementação do algoritmo

O algoritmo implementado foi dividido em três partes: Agente Cliente, Agente Coordenador e o algoritmo de processamento.

4.3.1 Agente coordenador

O Agente coordenador ao ser iniciado registra-se no DF (*Directory Facilitator*), também conhecido como serviço de páginas amarelas fornecido pelo *framework* JADE, que consiste em um serviço de registro centralizado cuja entradas associam a identificação do agente aos seus serviços. O registro no DF será utilizado para que os agentes clientes possam o encontrar. Após o registro o agente coordenador gera os dados que serão processados e inicia os comportamentos de distribuição de dados e de balanceamento de cargas. A figura 4.1 demonstra o processo de inicialização do agente coordenador.

Os métodos que fazem o controle dos dados já processados foram todos criados como *thread safe* para manter a consistência da distribuição dos mesmos, em vista que os comportamentos dos agentes no *framework* JADE são *threads* e são escalonados utilizando o algoritmo Round Robin pelo próprio *framework*.

Após os dados estarem prontos para serem distribuídos o agente coordenador ativa o comportamento que realiza a distribuição dos dados. Este comportamento será executado até que todos os dados sejam distribuídos. Ao ser executado, realiza-se uma busca no DF por agentes do tipo cliente, uma nova *thread* é criada visando aumentar o desempenho da distribuição e um conjunto de dados é enviado para cada cliente registrado no sistema. Assim que um novo cliente é conectado ao DF este também passará a receber os dados. A figura 4.2 ilustra o processo de distribuição de dados.

O comportamento de balanceamento de carga é iniciado ao receber uma notificação de nível

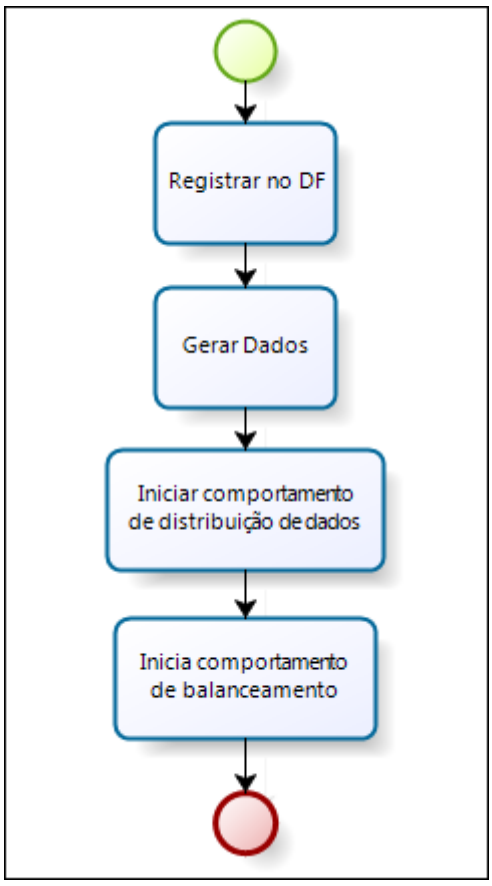


Figura 4.1: Inicialização do agente coordenador
(Próprio autor)

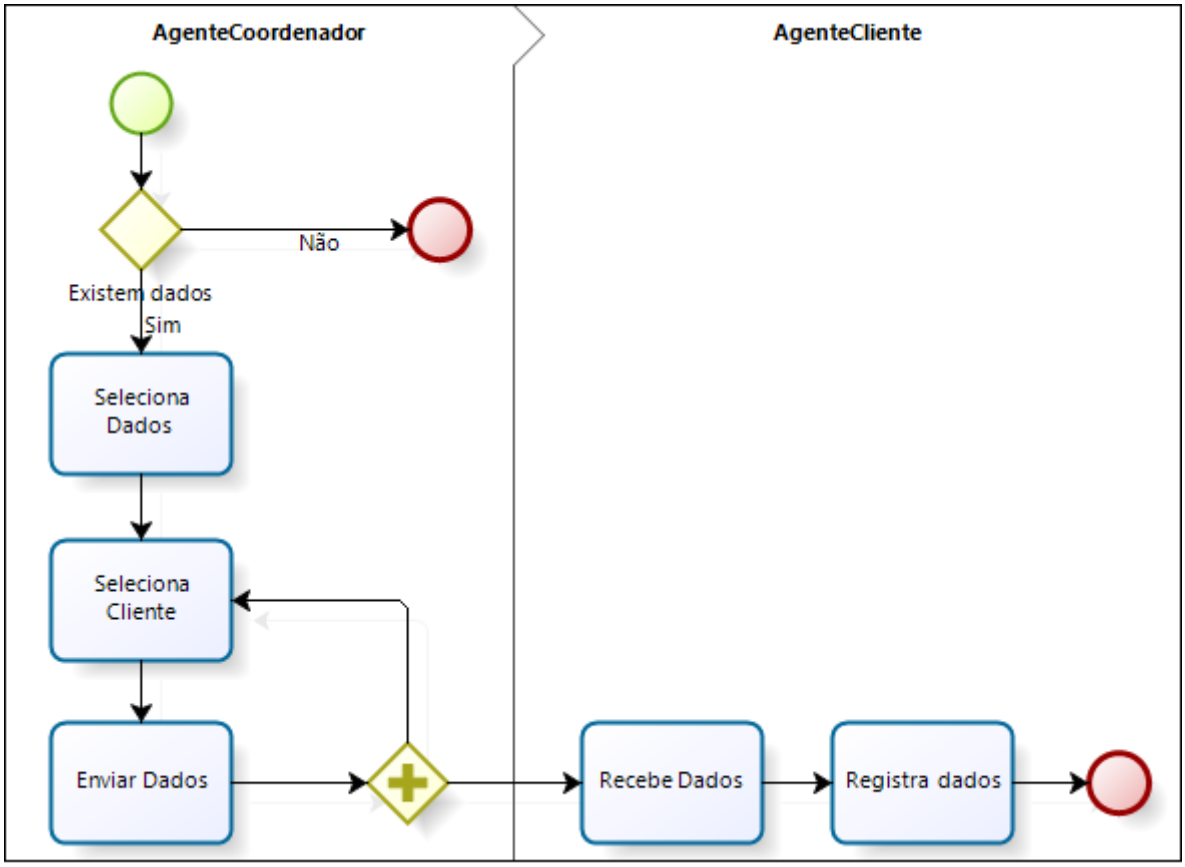


Figura 4.2: Distribuição dos dados pelo coordenador
(Próprio autor)

de carga de um agente cliente. Caso seja uma notificação de ociosidade o identificador do agente é registrado em uma lista no agente coordenador. Caso seja uma mensagem de notificação de sobrecarga o agente coordenador responde a mensagem com a identificação do agente que está a mais tempo ocioso para que a carga seja migrada.

4.3.2 Agente cliente

Cada agente quando iniciado registra-se no DF. O DF auxiliará na localização de agentes durante a execução. A figura 4.3 apresenta a inicialização do agente cliente.

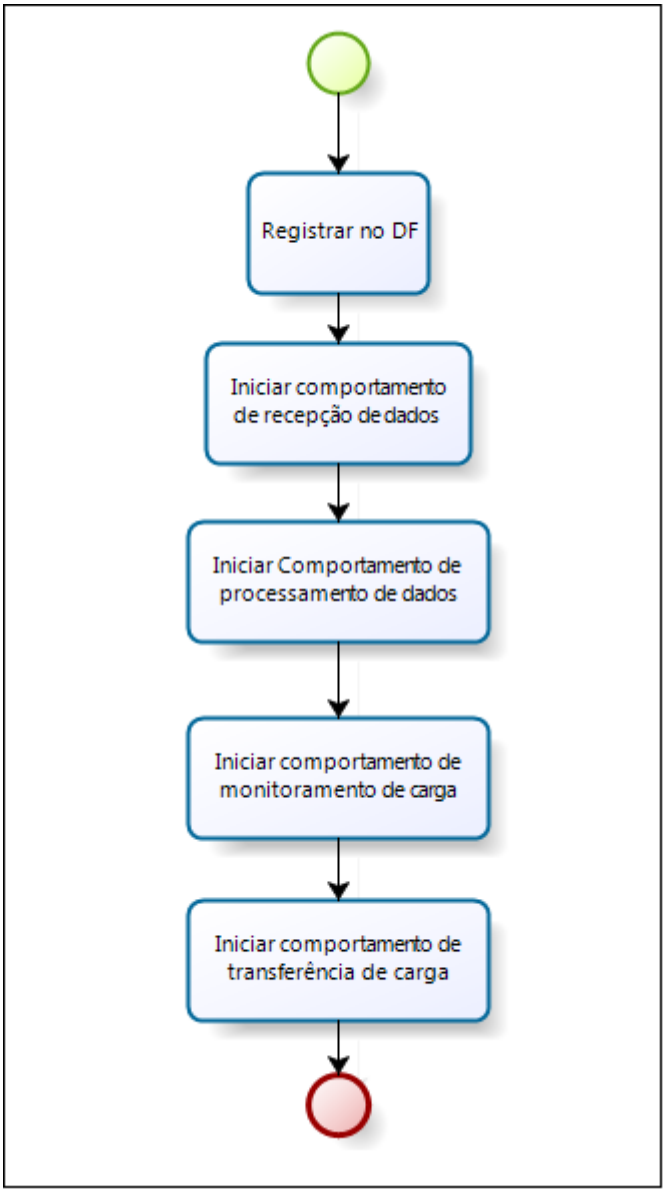


Figura 4.3: Inicialização do agente cliente
(Próprio autor)

Quando um agente cliente entra em execução é iniciado iniciado um comportamento cíclico de leitura de mensagem que tentará regatar sua mensagens em uma fila fornecida pelo *framework* JADE e caso esteja vazia o comportamento é bloqueado até que uma nova mensagem seja recebida. Este comportamento é responsável por receber os dados que deverão ser processados adicionando-os a uma lista.

O comportamento responsável por realizar o processamento também é cíclico e iniciado juntamente com o agente. Este verificará uma lista de dados a serem processados que é alimentada pelo comportamento que recebe as mensagens. Caso existam dados na fila é iniciado o processamento dos dados pelo dado que está a mais tempo na lista, caso contrário o comportamento é bloqueado aguardando o recebimento de novas mensagens. Este comportamento pode ser visto na figura 4.4.

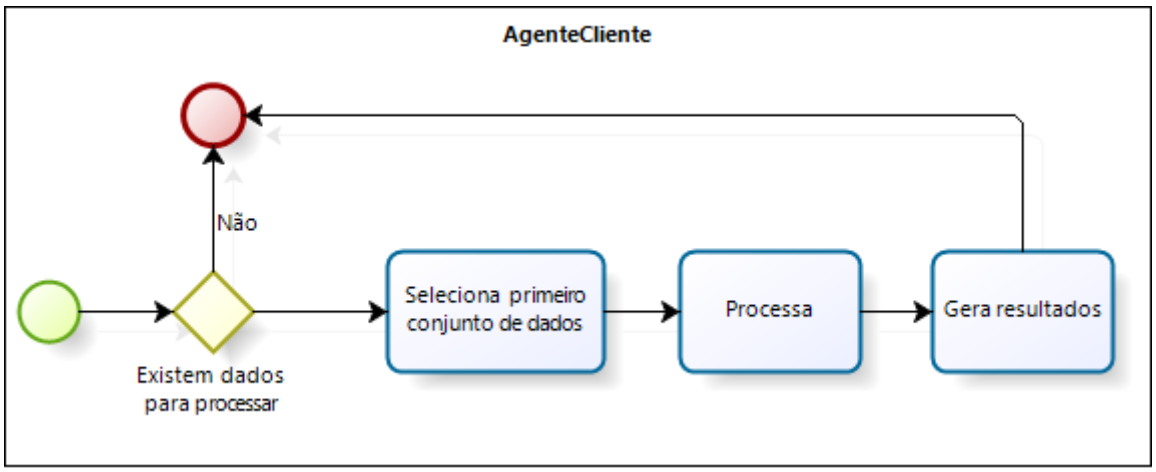


Figura 4.4: Processamento dos dados pelo Cliente
(Próprio autor)

Também foi criado um comportamento que irá monitorar a quantidade de dados a serem processados. Este comportamento é executado a cada 200 ms analisando o nível de carga do cliente, caso seja considerado ocioso ou sobrecarregado uma mensagem é enviada ao Agente Coordenador notificando a situação e o nível de carga para que este tome providências. Caso a lista esteja vazia o comportamento é bloqueado até que uma nova mensagem seja recebida. O fluxo do balanceamento de carga pode ser visto na figura 4.5.

Para a verificação do nível de carga foi estabelecido um limite máximo e um limite mínimo de quantidade de dados a serem processados. Caso o Agente Cliente possua mais dados do que o limite máximo permitido ele será considerado sobrecarregado, e caso esteja abaixo do limite de carga será considerado ocioso.

Para que os agentes cliente localizem o Agente coordenador é realizada um consulta no DF

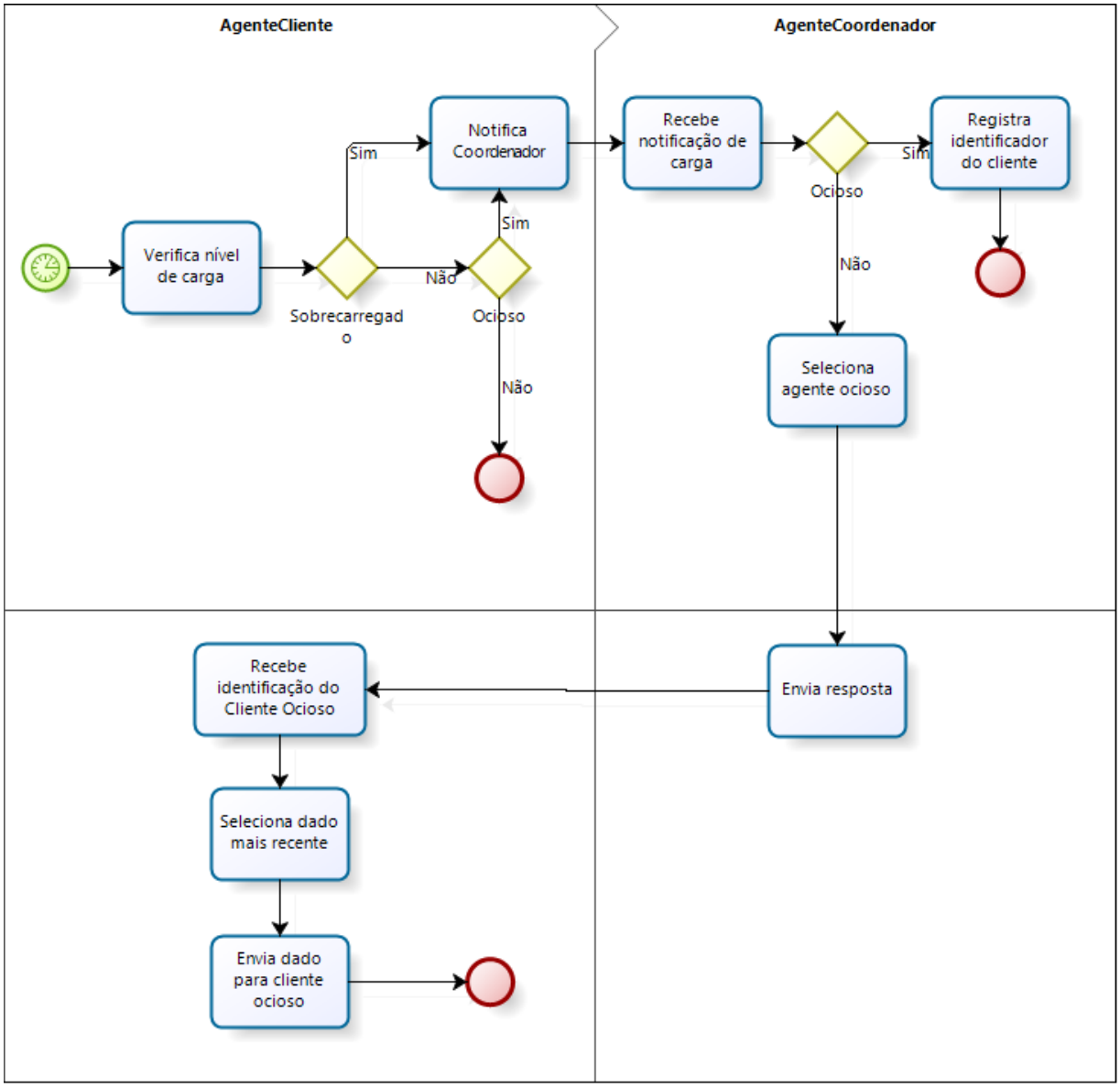


Figura 4.5: Processo de balanceamento de carga iniciado pelo cliente
(Próprio autor)

passando o tipo do agente desejado.

Foi criado um comportamento que é executado quando o cliente recebe uma notificação de movimentação de carga. Esta mensagem é recebida quando o cliente notifica o coordenador que seu nível de carga está acima do limite estabelecido, então o Servidor envia a identificação de um agente cliente ocioso. Ao receber esta mensagem o agente cliente sobrecarregado envia a última tarefa que foi recebida para o agente enviado pelo coordenador.

4.3.3 Algoritmo de processamento

Para realizar os testes foi implementado um algoritmo de verificação de números primos. A escolha deste algoritmo foi realizada devido ao diferente tempo de processamento de acordo com o número de entrada e também pela simplicidade de implementação, em vista que não era este o foco do projeto.

4.3.4 Geração dos dados para processamento

Para a geração dos dados foi utilizado o padrão de projeto *singleton* garantindo a existência de apenas uma instância da classe que fornece todos os dados que serão distribuídos em seu método construtor.

O algoritmo irá gerar uma determinada quantidade de números inteiros randomicamente populando uma lista que será distribuída pelo agente coordenador.

4.4 Resultados

Os experimentos foram realizados utilizando uma rede ethernet 100 Mbps interligados por um *switch* e compostos de 3 computadores clientes Pentium IV 2.8 GHz com 1 GB de Memória RAM DDR2 e um servidor AMD Athon II X2 2.1 GHz com 2 GHz de memória RAM DDR3.

Foram realizados testes com massas de dados diferentes e foram tiradas métricas das quantidades de vezes que os *hosts* ficaram ociosos, sobrecarregados e quantas trocas foram realizadas.

Para simular um ambiente onde exista diferença de cargas, a distribuição ocorre de forma aleatória após a geração dos dados pelo agente coordenador.

A figura 4.6 apresenta a relação dos resultados entre a quantidade de dados processados e as ocorrências de ociosidade, de sobrecarga e a quantidade de balanceamentos realizados.

Também são apresentado os índices utilizados para determinar *hosts* ociosos e sobrecarregados.

Dados	Ocorrências ociosidade	Ocorrências sobrecarga	Trocas de Carga	Nível de ociosidade	Nível de Sobrecarga
100000	36	116	33	5	10
300000	142	198	126	15	30
500000	142	696	101	25	50
700000	179	1043	114	35	70
1000000	215	1552	122	50	100

Figura 4.6: Níveis dos *hosts* em relação a quantidade de dados processados
(Próprio autor)

A figura 4.7 apresenta um gráfico resultante dos dados apresentados na figura 4.6.

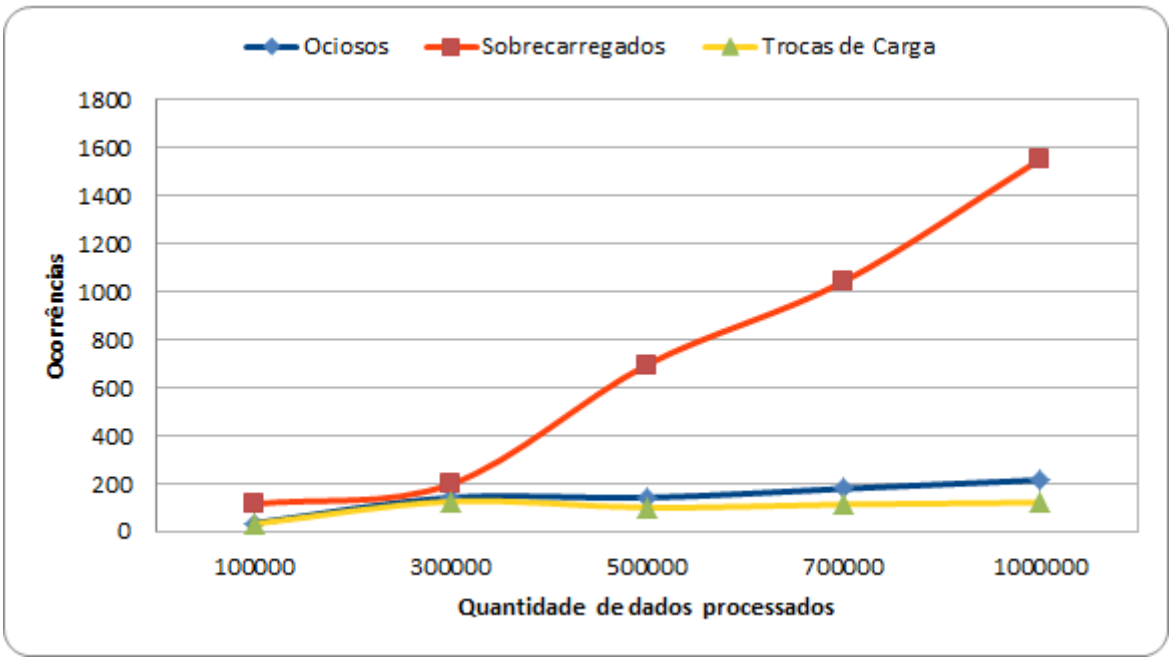


Figura 4.7: Gráfico representando os níveis dos *hosts* em relação a quantidade de dados processados
(Próprio autor)

Analisando o gráfico apresentado na figura 4.7, constata-se que conforme a quantidade de dados processados aumenta, existe um aumento significativo na quantidade de ocorrências de sobrecarga do sistema, sendo que os as trocas de carga ocorrem conforme a existência de *hosts* ociosos.

Também foram tiradas métricas de performance do sistema que realiza o balanceamento de carga e comparadas com o mesmo sistema que não aplica o balanceamento. Os resultados obtidos podem ser encontrados na figura 4.8.

Dados	Tempo com balanceamento (ms)	Tempo sem balanceamento (ms)
100000	6788	5592
300000	16059	15677
500000	24370	27834
700000	32174	35144
1000000	45180	63094

Figura 4.8: Tempo de execução em relação a quantidade de dados processados
(Próprio autor)

O gráfico obtido com estes resultados pode ser visualizado na figura 4.9.

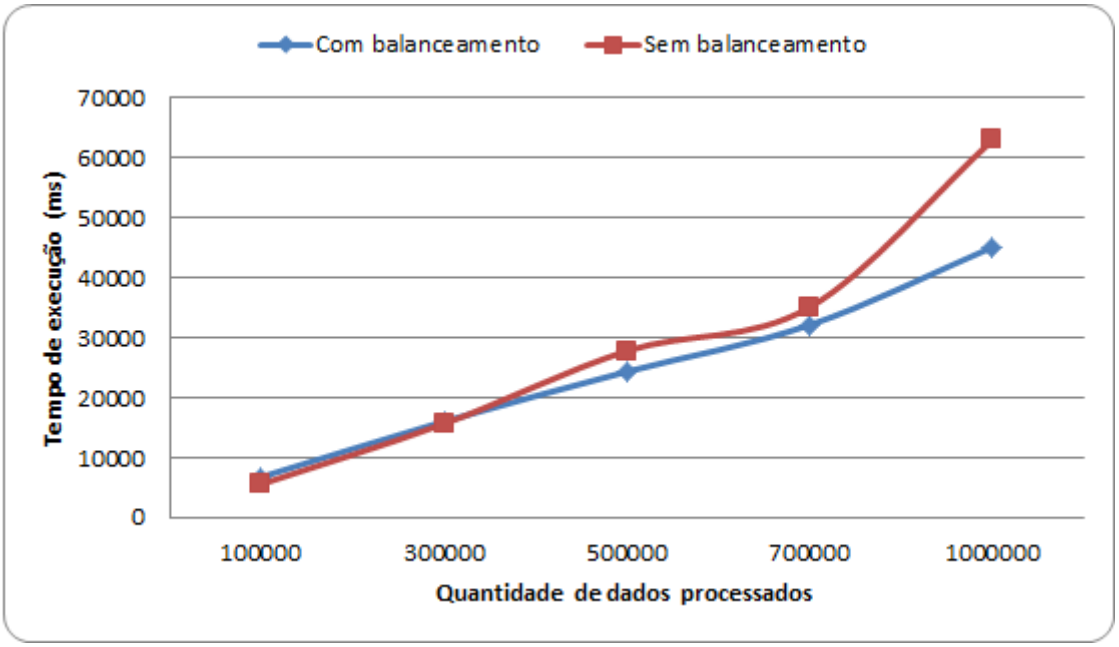


Figura 4.9: Gráfico que representa a relação entre o tempo de execução e a quantidade de dados os dados processados
(Próprio autor)

Analisando o gráfico apresentado na figura 4.9 pode observar-se que com poucos dados a serem processados o algoritmo que não aplica o balanceamento tende a ser mais eficiente, e conforme a massa de dados aumenta o tempo de execução do algoritmo que aplica o balanceamento tende a ter um melhor desempenho.

4.5 Conclusões

Objetivando uma melhoria no desempenho referente ao tempo de execução da aplicação distribuída, foram utilizados agentes reativos que analisam o ambiente ativando o balancea-

mento de carga conforme o necessário. Para implementação do sistema multiagente utilizou-se o *framework* JADE que abstraiu grande parte da troca de mensagens realizadas entre os agentes, da localização dos agentes de forma dinâmica, e aplicando o algoritmo de escalonamento *Round Robin* entre seus próprios comportamentos. Dessa forma o *framework* JADE foi suficiente para monitoramento, comunicação e localização de agentes.

Para a simulação de um ambiente onde a divisão das tarefas fossem de diferentes cargas, o agente coordenador distribui aleatoriamente as tarefas entres os agentes clientes. Para determinar o nível de ociosidade e sobrecarga foi levado em conta a quantidade de dados que deveriam ser processados, calculando uma porcentagem em cima deste valor e definindo o nível máximo e mínimo de carga. Para cálculo da ociosidade utilizou-se 0,00005% e para sobrecarga 0,0001%.

Para obtenção do nível de carga do agente foi criado um comportamento no cliente, onde cada um conhecia seu próprio nível e o agente coordenador recebe notificações quando o nível de ociosidade ou sobrecarga são atingidos, iniciando o processo de transferência de carga. Este comportamento mostrou-se eficiente ao efetuar o monitoramento dos agentes sem gerar um trafego prejudicial ao sistema. O resultado obtido com o monitoramento das cargas serviu como parâmetro para que o agente coordenador efetuasse as transferências de cargas, aliviando a sobrecarga de determinados *hosts*.

Ao analisar os resultados obtidos observou-se que quando aplicamos o balanceamento de cargas em sistemas heterogêneos é possível melhorar o desempenho em relação ao tempo de execução, desde que, o custo de manter *hosts* ociosos supere o tempo levado para monitorar e realocar tarefas. Dessa forma os sistemas multiagentes reativos mostraram-se eficientes na aplicação do balanceamento de cargas em sistemas distribuídos com grandes massas de dados.

Referências Bibliográficas

Alvares, Luis Otavio and Sichman, Jaime Simão. Introdução aos sistemas multiagentes. *XVII Congresso da SBC*, 1997.

BAKER, A. J. *A java-based agent framework for multiagent systems. Development and Implementation*. Tese (Doutorado) — Cincinnati: Department of Electrical & Computer Engineering and Computer Science University of Cincinnati, 1997.

BARRETO, J. M. *IAD*. Abril 2011. Disponível em: <http://www.inf.ufsc.br/˜ barreto/Projetos/Analucia/IAD.htm>.

BOND, A.; GASSER, L. *Readings in distributed artificial intelligence*. [S.l.]: M. Kaufmann, 1988. ISBN 9780934613637.

BRANCO, K. R. L. J. C. *Índice de Carga E Desempenho em Ambientes Paralelos/ Distribuídos - Modelagem e Métricas*. Tese (Doutorado) — ICMC-USP, 2004.

Brooks, R. A robust layered control system for a mobile robot. *IEEE Journal on Robotics and Automation*, IEEE, v. 2, n. 1, p. 14–23, mar. 1986. ISSN 0882-4967.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. *Sistemas Distribuídos - CONCEITOS E PROJETO*. 4. ed. [S.l.]: BOOKMAN COMPANHIA ED, 2007. ISBN 9788560031498.

DEMAZEAU, Y.; MÜLLER, J. *Decentralized A.I.: proceedings of the First European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Cambridge, England, August 16-18*. [S.l.]: North-Holland, 1990. (Decentralized A. I). ISBN 9780444887054.

GARCIA, A. C. B.; SICHMAN, J. S. Agentes e sistemas multiagentes. In: REZENDE, S. O. (Ed.). *Sistemas Inteligentes: Fundamentos e Aplicações*. Barueri, São Paulo, Brasil: Editora Manole Ltda., 2003.

HÜBNER, J.; BORDINI, R.; VIEIRA, R. Introdução ao desenvolvimento de sistemas multiagentes com jason. In: *XII Escola de Informática da SBC*. Guarapuava: UNICENTRO, 2004. v. 2, p. 51–89. Disponível em: <http://www.inf.furb.br/ jomi/pubs/2004/Hubner-eriPR2004.pdf>.

JENNINGS, N. et al. *Automated Negotiation*. Manchester, UK: [s.n.], 2000. 23-30 p.

MAES, P. Artificial Life Meets Entertainment: Lifelike Autonomous Agents. *ACM*, 1995.

MARQUES, G. M. Avaliação de Índices de carga e de desempenho em ambientes paralelos distribuídos com agentes móveis. *Centro Universitário Eurípedes de Marília - UNIVEM*, 2008.

MENESES, E. X. Integração de agentes de informação. *Jornada de Atualização em Inteligência Artificial*, 2001.

MULLENDER, S. J. *Distributed Systems*. 2. ed. [S.l.]: Addison-Wesley, 1993. ISBN 9788560031498.

REIS, L. P. *Coordenação em Sistemas Multi-Agente: Aplicações na Gestão Universitária e Futebol Robótico*. Tese (Doutorado) — Faculdade de Engenharia da Universidade do Porto, 2003.

REZENDE, S. O. *Sistemas Inteligentes: Fundamentos e Aplicações*. [S.l.]: Editora Manole Ltda., 2003. ISBN 9788520416839.

RUSSELL, S. J.; NORVIG, P. *Artificial intelligence: a modern approach*. [S.l.]: Prentice Hall, 2010. (Prentice Hall series in artificial intelligence). ISBN 9780136042594.

SHIVARATRI, N. G.; KRUEGER, P.; SINGHAL, M. Load Distribution for Locally Distributed Systems. *IEEE Computer*, 1992.

SICHMAN, J. S. *Du raisonnement social chez les agents: une approche fondée sur la théorie de la dépendance*. Tese (Doutorado) — Institut National Polytechnique de Grenoble, 1995.

SOUZA, E. M. S. de. Uma estrutura de agentes para assessoria na internet. *Universidade Federal de Santa Catarina, Florianópolis*, 1996.

TANENBAUM, A.; STEEN, M. *Distributed systems: principles and paradigms*. 2. ed. [S.l.]: Pearson Prentice Hall, 2007. ISBN 9780132392273.

TANENBAUM, A. S. *Modern Operating Systems*. 3. ed. [S.l.]: Pearson/Prentice Hall, 2009. ISBN 9780138134594.

WEISS, G. *Multiagent systems: a modern approach to distributed artificial intelligence*. [S.l.]: MIT Press, 1999. (Intelligent Robotics and Autonomous Agents). ISBN 9780262731317.

APÊNDICE A – Implementação

Código A.1: Implementação do agente coordenador

```
1  package Agentes;
2
3  import jade.core.AID;
4  import jade.core.Agent;
5  import jade.core.behaviours.CyclicBehaviour;
6  import jade.core.behaviours.SimpleBehaviour;
7  import jade.domain.DFService;
8  import jade.domain.FIPAAgentManagement.DFAgentDescription;
9  import jade.domain.FIPAAgentManagement.ServiceDescription;
10 import jade.domain.FIPAException;
11 import jade.lang.acl.ACLMessage;
12 import jade.lang.acl.MessageTemplate;
13 import jade.proto.SubscriptionInitiator;
14 import java.io.IOException;
15 import java.util.ArrayList;
16 import util.Carga;
17 import util.Dados;
18
19 /**
20  * Para executar o Jade:
21  * java -cp lib\jade.jar;build\classes jade.Boot -gui -agents
22  *    coordenador:Agentes.AgenteCoordenador
23  */
24 public class AgenteCoordenador extends jade.core.Agent {
25
26     private int ultimoDadoProcessado = 0;
27     public static final int DADOS_POR_PROCESSAMENTO = 5;
28     Dados dados;
29     ArrayList<Carga> agentesOciosos = new ArrayList<>();
30     ArrayList<Carga> agentesSobrecarregados = new ArrayList<>();
31     int ociosos = 0;
32     int sobrecarregados = 0;
```

```
33      int trocas = 0;
34      long time;
35
36      public AgenteCoordenador() {
37      }
38
39      protected void setup() {
40          time = System.currentTimeMillis();
41          DFAgentDescription dfd = new DFAgentDescription();
42          dfd.setName(getAID()); // Informamos a AID do agente
43
44          // criar a decricao do servico para registrar no DF
45          ServiceDescription sd = new ServiceDescription();
46          sd.setType("Coordenador");  // Tipo do Servico
47          sd.setName("Servico1"); //Nome do Servico
48          dfd.addServices(sd);
49          // Registrando no DF
50          try {
51              DFService.register(this, dfd);
52          } catch (FIPAException e) {
53              System.out.println("Agentes.AgenteCoordenador.setup() - Erro -
                    Nao foi possivel conectar ao DF: " + e.getMessage());
54          }
55
56          addBehaviour(new SimpleBehaviour(this) {
57
58              int cont = 0;
59
60              /* Envia Dados para Cliente Processar */
61              public void action() {
62                  ArrayList<AID> clientes = buscaClientes();
63
64                  if (!clientes.isEmpty() || clientes != null ) {
65                      for (final AID aid : clientes) {
66                          new Thread() {
67
68                              @Override
69                              public void run() {
70                                  ACLMessage msg = new ACLMessage(ACLMessage.
                                        INFORM);
71                                  msg.addReceiver(aid);
72                                  try {
```

```
73                                  msg.setContentObject(
                                        getDadosParaProcessar());
74                              } catch (IOException ex) {
75                                  System.out.println("Agentes.
                                        AgenteCoordenador::setup() - Erro no
                                         envio da mensagem: " + ex);
76                              }
77                              myAgent.send(msg);
78                          }
79                      }.start();
80                      cont++;
81                  }
82              } else {
83                  block();
84              }
85          }
86
87          @Override
88          public boolean done() {
89              if (ultimoDadoProcessado >= Dados.getInstance().TAMANHO) {
90                  return true;
91              } else {
92                  return false;
93              }
94          }
95
96          protected void takeDown() {
97              System.out.println("Todas as informacoes foram enviadas");
98          }
99      });
100
101
102     /* Recebe notificacao de carga */
103     addBehaviour(new CyclicBehaviour(this) {
104
105         MessageTemplate filtro = MessageTemplate.MatchConversationId("
                Ocioso");
106
107         public void action() {
108             ACLMessage msg = myAgent.receive(filtro);
109             if (msg != null) {
110                 int carga = Integer.parseInt(msg.getContent());
111                 if (carga > 0) {
```

```
112                         registraCarga(msg, carga);
113                         respondeSobrecarga(myAgent, msg);
114                     } else {
115                         registraCarga(msg, carga);
116                         respondeOciosidade(myAgent, msg);
117                     }
118                 } else {
119                     block();
120                 }
121             }
122         });
123 
124         /* Recebe notificacao de finalizacao de processamento do cliente */
125         addBehaviour(new CyclicBehaviour(this) {
126 
127             MessageTemplate filtro = MessageTemplate.MatchConversationId("
                    fim");
128 
129             public void action() {
130                 ACLMessage msg = myAgent.receive(filtro);
131                 if (msg != null) {
132                     System.out.println("Fim de lista: " + msg.getSender().
                            getLocalName());
133                     System.out.println("Ociosos (cont): " + ociosos);
134                     System.out.println("sobre (cont): " + sobrecarregados);
135                     System.out.println("Trocas (cont): " + trocas);
136                     System.out.println("ultimo: " + ultimoDadoProcessado);
137                     System.out.println("tamanho: " + Dados.getInstance().
                            TAMANHO);
138                     System.out.println("tempo total: " + (System.
                            currentTimeMillis() - time));
139                 }
140                 block();
141             }
142         });
143 
144     }
145 
146     protected ArrayList<AID> buscaClientes() {
147         DFAgentDescription busca = new DFAgentDescription();
148         ServiceDescription descricaoServico = new ServiceDescription();
149         descricaoServico.setType("Cliente");
150         busca.addServices(descricaoServico);
```

```
151
152         ArrayList<AID> listaClientes = new ArrayList<>();
153
154         DFAgentDescription[] resultados;
155         try {
156             resultados = DFService.search(this, busca);
157             for (DFAgentDescription resultado : resultados) {
158                 listaClientes.add(resultado.getName());
159             }
160         } catch (FIPAException ex) {
161             System.out.println("Agentes.AgenteCoordenador.buscaClientes() -
                    Erro - Falha ao conectar ao DF: " + ex.getMessage());
162         }
163
164         return listaClientes;
165     }
166
167     protected synchronized void registraCarga(ACLMessage msg, Integer carga
        ) {
168         if (carga > 0) {
169             sobrecarregados++;
170             agentesSobrecarregados.add(new Carga(msg.getSender(), carga));
171         } else {
172             ociosos++;
173             agentesOciosos.add(new Carga(msg.getSender(), carga));
174         }
175     }
176
177     protected synchronized void respondeSobrecarga(Agent myAgent,
        ACLMessage msg) {
178         ACLMessage resposta = msg.createReply();
179         try {
180             if (agentesOciosos.size() > 0) {
181                 trocas++;
182                 resposta.setContentObject(agentesOciosos.remove(0).
                        getAgente());
183                 myAgent.send(resposta);
184             }
185         } catch (IOException ex) {
186             System.out.println("Agentes.AgenteCoordenador::setup() - Erro
                ao setar conteudo da resposta da notificacao de carga (
                Sobrecarregado): " + ex);
187         }
```

```
188
189     }
190
191     protected synchronized void respondeOciosidade(Agent myAgent,
            ACLMessage msg) {
192         ACLMessage resposta = msg.createReply();
193
194         try {
195             if (agentesSobrecarregados.size() > 0) {
196                 resposta.setContentObject(agentesSobrecarregados.remove(0).
                        getAgente());
197                 myAgent.send(resposta);
198             }
199         } catch (IOException ex) {
200             System.out.println("Agentes.AgenteCoordenador::setup() - Erro
                    ao setar conteudo da resposta da notificacao de carga (
                    Ocioso): " + ex);
201         }
202     }
203
204     public synchronized ArrayList<Integer> getDadosParaProcessar() {
205         Integer quantidadeDados = new Integer((int) (1 + Math.random() *
                1000));
206         ArrayList<Integer> d = Dados.getInstance().getDados(
                ultimoDadoProcessado, quantidadeDados);
207         ultimoDadoProcessado += quantidadeDados;
208         return d;
209     }
210
211     protected void takeDown() {
212         try {
213             DFService.deregister(this);
214         } catch (FIPAException e) {
215             System.out.println("Agentes.AgenteCoordenador.takeDown() - Erro
                        - Erro ao finalizar registro no DF " + e.getMessage());
216         }
217     }
218 }
```

Código A.2: Implementação do agente cliente

```
1 package Agentes;
2
3 import Algoritmo.NumeroPrimo;
```

```
4  import jade.core.AID;
5  import jade.core.Agent;
6  import jade.core.behaviours.CyclicBehaviour;
7  import jade.domain.DFService;
8  import jade.domain.FIPAAgentManagement.DFAgentDescription;
9  import jade.domain.FIPAAgentManagement.ServiceDescription;
10 import jade.domain.FIPAException;
11 import jade.lang.acl.ACLMessage;
12 import jade.lang.acl.MessageTemplate;
13 import jade.lang.acl.UnreadableException;
14 import java.io.IOException;
15 import java.util.ArrayList;
16 import util.Dados;
17 import util.Resposta;
18
19 public class AgenteCliente extends jade.core.Agent {
20
21     final int NIVEL_SOBRECARGA = (int) (Dados.getInstance().TAMANHO *
          0.0001);
22     final int NIVEL_OCIOSIDADE = (int) (Dados.getInstance().TAMANHO *
          0.00005);
23     ArrayList<ArrayList<Integer>> listaDados;
24     ArrayList<ArrayList<Resposta>> listaRespostas;
25     ArrayList<Integer> dado;
26     long tempo;
27     int ocioso;
28     int sobre;
29
30     public AgenteCliente() {
31         listaDados = new ArrayList<>();
32         listaRespostas = new ArrayList<>();
33         dado = new ArrayList<>();
34     }
35
36     protected void setup() {
37         tempo = System.currentTimeMillis();
38         DFAgentDescription dfd = new DFAgentDescription();
39         dfd.setName(getAID()); // Informamos a AID do agent
40
41         // criar a decricao do servico para registrar no DF
42         ServiceDescription sd = new ServiceDescription();
43         sd.setType("Cliente");  // Tipo do Servico
44         sd.setName("Servico2"); //Nome do Servico
```

```
45          dfd.addServices(sd);
46          // Registrando no DF
47          try {
48              DFService.register(this, dfd);
49          } catch (FIPAException e) {
50              System.out.println("AgenteCliente::setup() - Erro ao conectar
                    ao DF: " + e.getMessage());
51          }
52
53          /* RecebeMensagem */
54          addBehaviour(new CyclicBehaviour(this) {
55              MessageTemplate filtro = MessageTemplate.not(MessageTemplate.
                    MatchConversationId("Ocioso"));
56
57              public void action() {
58                  int i = 0;
59                  boolean temMensagem = true;
60                  while (i <= 5 && temMensagem) {
61                      ACLMessage msg = receive(filtro);
62                      if (msg != null) {
63                          try {
64                              ArrayList<Integer> dado = (ArrayList<Integer>)
                                    msg.getContentObject();
65                              listaDados.add(dado);
66                          } catch (UnreadableException ex) {
67                              System.out.println("Agentes.AgenteCliente.setup
                                    () - Erro - RecebeMsg::Erro ao receber dados
                                    " + ex.getMessage());
68                          }
69                      } else {
70                          temMensagem = false;
71                      }
72                      i++;
73                  }
74
75                  if (!temMensagem) {
76                      block(500);
77                  }
78              }
79          });
80
81          /*
82           * Comportamento de processar se os numeros de uma lista sao primos
```

```
83          */
84         addBehaviour(new CyclicBehaviour(this) {
85             @Override
86             public void action() {
87                 if (listaDados.isEmpty()) {
88                     block();
89                 } else {
90                     ArrayList<Resposta> respostas = processarDados(
                           listaDados.remove(0));
91                     listaRespostas.add(respostas);
92                 }
93             }
94         });
95 
96         /*
97          * Monitoramento para balanceamento executado a cada 200 ms
98          */
99         addBehaviour(new CyclicBehaviour(this) {
100 
101             int vezesOcioso = 0;
102 
103             @Override
104             public void action() {
105                 if (listaDados.size() >= NIVEL_SOBRECARGA) {
106                     sobre++;
107                     notificarCarga(myAgent, listaDados.size());
108                 } else if (!listaDados.isEmpty() && listaDados.size() <=
                        NIVEL_OCIOSIDADE) {
109                     ocioso++;
110                     notificarCarga(myAgent, -listaDados.size());
111                 }
112 
113                 if (listaDados.isEmpty()) {
114                     System.out.println("Monitora: Lista vazia finalizando:
                            ("+(System.currentTimeMillis()-tempo)+" ms)");
115                     System.out.println("Sobre: "+sobre);
116                     System.out.println("Ocioso: "+ocioso);
117                     System.out.println("Nivel Sobrecarga: "+
                            NIVEL_SOBRECARGA);
118                     System.out.println("Nivel Ociosidade: "+
                            NIVEL_OCIOSIDADE);
119                     notificarFimCoordenador(myAgent);
120                     block();
```

```
121                 } else {
122                     block(2000);
123                 }
124             }
125         });
126 
127         /*
128          * Recebe requisicao para transferencia de carga
129          */
130         addBehaviour(new CyclicBehaviour(this) {
131 
132             MessageTemplate filtro = MessageTemplate.MatchConversationId("
                    Ocioso");
133 
134             @Override
135             public void action() {
136                 ACLMessage msg = myAgent.receive(filtro);
137                 if (msg != null) {
138                     AID cliente = null;
139                     try {
140                         cliente = (AID) msg.getContentObject();
141                     } catch (UnreadableException ex) {
142                         System.out.println("Agentes.AgenteCliente.setup() -
                                Erro - Falha ao ler destinatario dos dados (
                                Recebe Requisicao): " + ex.getMessage());
143                     }
144                     if (cliente != null && listaDados.size() > 1) {
145                         ArrayList<Integer> dado = listaDados.remove(
                                listaDados.size() - 1);
146                         ACLMessage mensagem = new ACLMessage(ACLMessage.
                                INFORM);
147                         try {
148                             mensagem.addReceiver(cliente);
149                             mensagem.setContentObject(dado);
150                         } catch (IOException ex) {
151                             System.out.println("Agentes.AgenteCliente.setup
                                    () - Erro - Falha ao converter dados (Recebe
                                     Requisicao): " + ex.getMessage());
152                         }
153                         myAgent.send(mensagem);
154                     } else {
155                         block();
156                     }
```

```
157                 }
158             }
159         });
160     }
161
162     protected ArrayList<Resposta> processarDados(ArrayList<Integer> dados)
        {
163         NumeroPrimo numeroPrimo = new NumeroPrimo();
164         ArrayList<Resposta> respostas = new ArrayList<>();
165         for (Integer d : dados) {
166             respostas.add(new Resposta(d, numeroPrimo.isPrimo(d)));
167         }
168         return respostas;
169     }
170
171     protected void notificarCarga(Agent emissor, int nivelCarga) {
172         ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
173         msg.addReceiver(buscaCoordenador());
174         msg.setConversationId("Ocioso");
175         msg.setContent(String.valueOf(nivelCarga));
176         emissor.send(msg);
177     }
178
179     protected void notificarFimCoordenador(Agent emissor) {
180         ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
181         msg.addReceiver(buscaCoordenador());
182         msg.setConversationId("fim");
183         msg.setContent("fim");
184         emissor.send(msg);
185     }
186
187     protected AID buscaCoordenador() {
188         DFAgentDescription busca = new DFAgentDescription();
189         ServiceDescription descricaoServico = new ServiceDescription();
190         descricaoServico.setType("Coordenador");
191         busca.addServices(descricaoServico);
192
193         AID coordenador = new AID();
194         try {
195             coordenador = DFService.search(this, busca)[0].getName();
196         } catch (FIPAException ex) {
197             System.out.println("Agentes.AgenteCliente.buscaCoordenador() -
                Erro - Falha ao conectar ao DF: " + ex.getMessage());
```

```
198             }
199
200             return coordenador;
201         }
202
203         protected void takeDown() {
204             try {
205                 DFService.deregister(this);
206             } catch (FIPAException e) {
207                 System.out.println("Agentes.AgenteCliente.takeDown() - Erro -
                        Erro ao finalizar registro no DF " + e.getMessage());
208             }
209         }
210 }
```

Código A.3: Implementação do agente cliente

```
1  package Agentes;
2
3  import Algoritmo.NumeroPrimo;
4  import jade.core.AID;
5  import jade.core.Agent;
6  import jade.core.behaviours.CyclicBehaviour;
7  import jade.domain.DFService;
8  import jade.domain.FIPAAgentManagement.DFAgentDescription;
9  import jade.domain.FIPAAgentManagement.ServiceDescription;
10 import jade.domain.FIPAException;
11 import jade.lang.acl.ACLMessage;
12 import jade.lang.acl.MessageTemplate;
13 import jade.lang.acl.UnreadableException;
14 import java.io.IOException;
15 import java.util.ArrayList;
16 import util.Dados;
17 import util.Resposta;
18
19 public class AgenteCliente extends jade.core.Agent {
20
21     final int NIVEL_SOBRECARGA = (int) (Dados.getInstance().TAMANHO *
          0.0001);
22     final int NIVEL_OCIOSIDADE = (int) (Dados.getInstance().TAMANHO *
          0.00005);
23     ArrayList<ArrayList<Integer>> listaDados;
24     ArrayList<ArrayList<Resposta>> listaRespostas;
25     ArrayList<Integer> dado;
```

```
26      long tempo;
27      int ocioso;
28      int sobre;
29
30      public AgenteCliente() {
31          listaDados = new ArrayList<>();
32          listaRespostas = new ArrayList<>();
33          dado = new ArrayList<>();
34      }
35
36      protected void setup() {
37          tempo = System.currentTimeMillis();
38          DFAgentDescription dfd = new DFAgentDescription();
39          dfd.setName(getAID()); // Informamos a AID do agent
40
41          // criar a decricao do servico para registrar no DF
42          ServiceDescription sd = new ServiceDescription();
43          sd.setType("Cliente");  // Tipo do Servico
44          sd.setName("Servico2"); //Nome do Servico
45          dfd.addServices(sd);
46          // Registrando no DF
47          try {
48              DFService.register(this, dfd);
49          } catch (FIPAException e) {
50              System.out.println("AgenteCliente::setup() - Erro ao conectar
                    ao DF: " + e.getMessage());
51          }
52
53          /* RecebeMensagem */
54          addBehaviour(new CyclicBehaviour(this) {
55              MessageTemplate filtro = MessageTemplate.not(MessageTemplate.
                    MatchConversationId("Ocioso"));
56
57              public void action() {
58                  int i = 0;
59                  boolean temMensagem = true;
60                  while (i <= 5 && temMensagem) {
61                      ACLMessage msg = receive(filtro);
62                      if (msg != null) {
63                          try {
64                              ArrayList<Integer> dado = (ArrayList<Integer>)
                                    msg.getContentObject();
65                              listaDados.add(dado);
```

```
66                      } catch (UnreadableException ex) {
67                          System.out.println("Agentes.AgenteCliente.setup
                                () - Erro - RecebeMsg::Erro ao receber dados
                                " + ex.getMessage());
68                      }
69                  } else {
70                      temMensagem = false;
71                  }
72                  i++;
73              }
74
75              if (!temMensagem) {
76                  block(500);
77              }
78          }
79      });
80
81      /*
82       * Comportamento de processar se os numeros de uma lista sao primos
83       */
84      addBehaviour(new CyclicBehaviour(this) {
85          @Override
86          public void action() {
87              if (listaDados.isEmpty()) {
88                  block();
89              } else {
90                  ArrayList<Resposta> respostas = processarDados(
                        listaDados.remove(0));
91                  listaRespostas.add(respostas);
92              }
93          }
94      });
95
96      /*
97       * Monitoramento para balanceamento executado a cada 200 ms
98       */
99      addBehaviour(new CyclicBehaviour(this) {
100
101         int vezesOcioso = 0;
102
103         @Override
104         public void action() {
105             if (listaDados.size() >= NIVEL_SOBRECARGA) {
```

```
106                     sobre++;
107                     notificarCarga(myAgent, listaDados.size());
108                 } else if (!listaDados.isEmpty() && listaDados.size() <=
                        NIVEL_OCIOSIDADE) {
109                     ocioso++;
110                     notificarCarga(myAgent, -listaDados.size());
111                 }
112 
113                 if (listaDados.isEmpty()) {
114                     System.out.println("Monitora: Lista vazia finalizando:
                            ("+(System.currentTimeMillis()-tempo)+" ms)");
115                     System.out.println("Sobre: "+sobre);
116                     System.out.println("Ocioso: "+ocioso);
117                     System.out.println("Nivel Sobrecarga: "+
                            NIVEL_SOBRECARGA);
118                     System.out.println("Nivel Ociosidade: "+
                            NIVEL_OCIOSIDADE);
119                     notificarFimCoordenador(myAgent);
120                     block();
121                 } else {
122                     block(2000);
123                 }
124             }
125         });
126 
127         /*
128          * Recebe requisicao para transferencia de carga
129          */
130         addBehaviour(new CyclicBehaviour(this) {
131 
132             MessageTemplate filtro = MessageTemplate.MatchConversationId("
                    Ocioso");
133 
134             @Override
135             public void action() {
136                 ACLMessage msg = myAgent.receive(filtro);
137                 if (msg != null) {
138                     AID cliente = null;
139                     try {
140                         cliente = (AID) msg.getContentObject();
141                     } catch (UnreadableException ex) {
142                         System.out.println("Agentes.AgenteCliente.setup() -
                                Erro - Falha ao ler destinatario dos dados (
```

```
                        Recebe Requisicao): " + ex.getMessage());
143                 }
144                 if (cliente != null && listaDados.size() > 1) {
145                     ArrayList<Integer> dado = listaDados.remove(
                            listaDados.size() - 1);
146                     ACLMessage mensagem = new ACLMessage(ACLMessage.
                            INFORM);
147                     try {
148                         mensagem.addReceiver(cliente);
149                         mensagem.setContentObject(dado);
150                     } catch (IOException ex) {
151                         System.out.println("Agentes.AgenteCliente.setup
                                () - Erro - Falha ao converter dados (Recebe
                                 Requisicao): " + ex.getMessage());
152                     }
153                     myAgent.send(mensagem);
154                 } else {
155                     block();
156                 }
157             }
158         }
159     });
160 }
161
162 protected ArrayList<Resposta> processarDados(ArrayList<Integer> dados)
        {
163     NumeroPrimo numeroPrimo = new NumeroPrimo();
164     ArrayList<Resposta> respostas = new ArrayList<>();
165     for (Integer d : dados) {
166         respostas.add(new Resposta(d, numeroPrimo.isPrimo(d)));
167     }
168     return respostas;
169 }
170
171 protected void notificarCarga(Agent emissor, int nivelCarga) {
172     ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
173     msg.addReceiver(buscaCoordenador());
174     msg.setConversationId("Ocioso");
175     msg.setContent(String.valueOf(nivelCarga));
176     emissor.send(msg);
177 }
178
179 protected void notificarFimCoordenador(Agent emissor) {
```

```
180             ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
181             msg.addReceiver(buscaCoordenador());
182             msg.setConversationId("fim");
183             msg.setContent("fim");
184             emissor.send(msg);
185         }
186
187         protected AID buscaCoordenador() {
188             DFAgentDescription busca = new DFAgentDescription();
189             ServiceDescription descricaoServico = new ServiceDescription();
190             descricaoServico.setType("Coordenador");
191             busca.addServices(descricaoServico);
192
193             AID coordenador = new AID();
194             try {
195                 coordenador = DFService.search(this, busca)[0].getName();
196             } catch (FIPAException ex) {
197                 System.out.println("Agentes.AgenteCliente.buscaCoordenador() -
                        Erro - Falha ao conectar ao DF: " + ex.getMessage());
198             }
199
200             return coordenador;
201         }
202
203         protected void takeDown() {
204             try {
205                 DFService.deregister(this);
206             } catch (FIPAException e) {
207                 System.out.println("Agentes.AgenteCliente.takeDown() - Erro -
                        Erro ao finalizar registro no DF " + e.getMessage());
208             }
209         }
210 }
```

Código A.4: Implementação do gerador de dados

```
1  package util;
2
3  import java.util.ArrayList;
4
5  public class Dados {
6      private static Dados instancia = null;
7      public final int TAMANHO = 300_000;
8      public final int VALOR_MAXIMO_DADO = 1999999999;
```

```
9       ArrayList<Integer> vetor;
10
11      private Dados() {
12          vetor = new ArrayList<>();
13          for (int i=0; i < TAMANHO; i++) {
14              Integer valor = new Integer((int)(1 + Math.random() *
                    VALOR_MAXIMO_DADO));
15              vetor.add(valor);
16          }
17      }
18
19      public static synchronized Dados getInstance() {
20          if (instancia == null) {
21              instancia = new Dados();
22          }
23          return instancia;
24      }
25
26
27      public ArrayList<Integer> getDados(int offset, int limit) {
28          ArrayList<Integer> dados = new ArrayList<>();
29
30          limit = offset + limit;
31          if (limit >= vetor.size()) {
32              limit = vetor.size();
33          }
34
35          for (int i = offset; i < limit; i++ ) {
36              dados.add(vetor.get(i));
37          }
38          return dados;
39      }
40  }
```

Código A.5: Implementação da estrutura de controle de carga

```
1   package util;
2
3   import jade.core.AID;
4
5   public class Carga {
6       private long tempo = System.currentTimeMillis();
7           private AID agente;
8       private int carga;
```

```
9
10      public Carga(AID agente, int carga) {
11          this.agente = agente;
12          this.carga = carga;
13      }
14
15      public AID getAgente() {
16          return agente;
17      }
18
19      public int getCarga() {
20          return carga;
21      }
22
23      public long getTempo() {
24          return tempo;
25      }
26
27
28  }
```

Código A.6: Implementação da estrutura de armazenamento de resultados

```
1  package util;
2
3  public class Resposta {
4      public Integer numero;
5      public Boolean isPrimo;
6
7      public Resposta(Integer dado, Boolean isPrimo) {
8          this.numero = dado;
9          this.isPrimo = isPrimo;
10     }
11 }
```

Código A.7: Implementação do algoritmo de verificação de números primos

```
1  package Algoritmo;
2
3  import java.math.BigInteger;
4
5  public class NumeroPrimo {
6
7      public boolean isPrimo(int numero) {
```

```
8           if (!(numero % 2 == 0) && numero > 2) {
9               int i = 3;
10              while (i < Math.sqrt(numero)) {
11                  if (numero % i == 0) {
12                      return false;
13                  }
14                  i++;
15              }
16              return true;
17          }
18          return false;
19      }
20  }
```