UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

PROTÓTIPO DE SOFTWARE PARA CONSULTA DE BASE DE DADOS XML ATRAVÉS DE PROCESSAMENTO DISTRIBUÍDO USANDO AGENTES MÓVEIS EM AMBIENTE PEER-TO-PEER (P2P)

ALEXANDRE HELFRICH

BLUMENAU
2005

2005/1-01

ALEXANDRE HELFRICH

PROTÓTIPO DE SOFTWARE PARA CONSULTA DE BASE DE DADOS XML ATRAVÉS DE PROCESSAMENTO DISTRIBUÍDO USANDO AGENTES MÓVEIS EM AMBIENTE PEER-TO-PEER (P2P)

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciências da Computação — Bacharelado.

Prof. Paulo Fernando da Silva - Orientador.

BLUMENAU
2005

2005/1-01

PROCESSAMENTO DISTRIBUÍDO EM AMBIENTE P2P

(PEER-TO-PEER)

Por

ALEXANDRE HELFRICH

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Paulo Fernando da Silva– Orientador, FURB
Membro:	Prof. Francisco Adell Péricas – FURB
Membro:	Prof. Jomi Fred Hubner– FURB

Blumenau, 14 de Junho de 2005

Dedico este trabalho a minha esposa Marien Burghardt Helfrich e filho Matheus Helfrich, que junto comigo suportaram as restrições provenientes do tempo que passei estudando e pesquisando na faculdade, que entenderam a minha ausência principalmente para o desenvolvimento deste trabalho de conclusão do curso.

AGRADECIMENTOS

À Deus, pelo seu imenso amor e graça.

À minha esposa e filho por sua compreensão.

À minha família, que sempre me apoiou e incentivou a continuar.

Aos meus amigos, pelos momentos compartilhados, dentro e fora da faculdade.

Ao meu orientador, professor Paulo Fernando da Silva, por ter acreditado na minha capacidade de conclusão deste trabalho.

Em especial aos professores José Roque Voltolini da Silva, Jomi Fred Hübner, ao professor Paulo César Rodacki Gomes, Joyce Martins e Mauricio Capobianco Lopes por suas valiosas dicas.

À todos os demais professores que contribuíram para minha formação.

À todas as pessoas que apesar de não estarem citadas aqui, contribuíram de alguma forma para a conclusão deste.

Os bons livros fazem “sacar” para fora o que a pessoa tem de melhor dentro dela.

Lina Sotis Francesco Moratti

RESUMO

Este trabalho apresenta a especificação e implementação de um protótipo de software que, utilizando agentes móveis para realizar processamento distribuído em ambiente *peer-to-peer* e pesquisa informações em arquivos XML em uma rede de computadores. Na implementação dos agentes móveis foi utilizada a linguagem Java. Este trabalho apresenta ainda os conceitos de utilização de arquivos XML como base de dados e sua funcionalidade. Os agentes móveis foram implementados na plataforma Aglets. Como resultado obtiveram dois protótipos de software, um denominado servidor de nomes, que gerência e distribui a lista de *peers* e outro denominado *peer*, encarregado de distribuir a tarefa de pesquisar em arquivos XML entre todos os *peers* da lista do servidor de nomes.

Palavras chaves: Agentes móveis. Mobilidade. Processamento distribuído. Base de dados XML.

ABSTRACT

This work presents the specification and implementation of a software archetype that using mobile agents to carry through processing distributed in a peer-to-peer environment, searchs for information in archives XML in a computer network. In the implementation of the mobile agents the Java language was used. This work also presents the concepts of use of archives XML as database and its functionality. The mobile agents had been implemented in the Aglets platform. As result it got two archetypes of software, one called name server manages and distributes that a list of peers, another called peer, in charge distributing the task to search in archives XML among all peers of the list of the name server.

Words keys: Mobile agents. Mobility. Distributed processing. XML database.

LISTA DE ILUSTRAÇÕES

Figura 1 - Ciclo de vida de um *aglet*. 23
Quadro 1 - Exemplo de um documento XML 25
Figura 2 - Diagrama de caso de uso do aplicativo Peer 29
Figura 3 - Diagrama de classes do protótipo do aplicativo *peer*. 30
Figura 4 - Diagrama de seqüência do caso de uso *InicializaPeer* do *AgletPeer*. 31
Figura 5 - Diagrama de seqüência dos casos de uso *CriarConsulta*. 32
Figura 6 - diagrama de seqüência do caso de uso *VisualizaResultado*. 34
Figura 7 - diagrama de seqüência do caso de uso *VisualizaBasesDisponiveis*. 34
Figura 8 – Diagrama de casos de uso do aplicativo Servidor de nomes. 35
Figura 9 – Diagrama de classes do aplicativo servidor de nomes. 36
Figura 10 – Diagrama de seqüência do caso de uso *VisualizaPeer* 37
Figura 11 – Diagrama de seqüência do caso de uso *InserePeer*. 38
Figura 12 – Diagrama de seqüência do caso de uso *excluiPeer*. 38
Figura 13 – Diagrama de seqüência do caso de uso *onDisposing*. 39
Quadro 2 –Importação da AAPI para utilização da classe *Aglet* 42
Quadro 3 – Instanciação dos objetos 43
Quadro 4 –Comunicação simples do *Peer* com o *AgletMsg*. 44
Quadro 5 – Utilização do *Subscribemessage* 45
Quadro 6 – Envio da mensagem *multicast*. 45
Quadro 7 – Pedido de conexão do *peer* ao servidor de nomes. 46
Quadro 8 – Pedido de atualização da lista de *peers* do *peer*. 47
Quadro 9 – Inserção do *peer* na tabela de *peers* do servidor de nomes. 47
Quadro 10 – Atualização da lista de *peers* utilizada nos *peers*. 48
Quadro 11 – Uso do evento *onDisposing* no servidor. 49
Quadro 12 – Utilização do evento *handleMessage* no servidor. 49
Quadro 13 – método *VerificaArquivos*(). 50
Quadro 14 – Implementação da classe *Carro*. 51
Quadro 15 – Implementação do *parser* dos arquivos XML. 52
Quadro 16 – Implementação da instanciação dos elementos do arquivo XML 53
Quadro 17 – Estrutura do arquivo XML. 53
Quadro 18 – Processamento do elemento do arquivo XML 54

Quadro 19 – Implementação da classe *listacarros.* 55
Quadro 20 – Base de dados de *atp:Matheus:13000/,* em forma de texto. 56
Quadro 21 – Base de dados de *atp://Alexandre:12000/,* em forma de texto. 57
Quadro 22 – Apresentação do aplicativo *Tahiti.* 58
Quadro 23 – Seleção do *aglet* a ser executado através do *Tahiti*. 59
Quadro 24 – Execução do servidor de nomes. 59
Quadro 25 – Apresentação do servidor de nomes com 2 *peers* conectados. 60
Quadro 26 – Visualização dos aplicativos *peers*. 60
Quadro 27 – Visualização da tela de pesquisa. 61
Quadro 28 – Visualização do resultado da pesquisa nos *peers*. 61

LISTA DE TABELAS E GRÁFICOS

Tabela 1 – Comparativo dos resultados em milisegundos. 62
Gráfico 1 – Comparativo de tempos gastos por registros. 63
Gráfico 2 – Diferença do tempo total dos casos de teste. 64

LISTA DE SIGLAS

AM – Agente Móvel ou *Móbile Agent*

API – *Aplication Programming Interface*

HTML – Linguagem de Marcação de Hipertexto ou *Hipertext Markup Language*

JVM – Máquina Virtual Java ou *Java Virtual Machine*

MAS – Sistemas de Agentes Móveis ou *Mobile Agent Systems*

P2P – Par a Par ou *peer-to-peer*

PCS – Computadores Pessoais ou *Personal Computers*

SGML – Modelo Generalizado de Linguagem de Marcação ou *Standard Generalized Markup Language*

XML – Linguagem de Marcação Extensível ou *extensible markup language*

SUMÁRIO

1 INTRODUÇÃO 12
1.1 OBJETIVOS DO TRABALHO 15
1.2 ESTRUTURA DO TRABALHO 16
2 FUNDAMENTAÇÃO TEÓRICA 17
2.1 REDES PEER-TO-PEER 17
2.2 AGENTES MÓVEIS 19
2.2.1 AGLETS 21
2.3 XML (EXTENSIBLE MARKUP LANGUAGE) 23
2.4 TRABALHOS CORRELATOS 25
3 DESENVOLVIMENTO DO PROTÓTIPO 27
3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO 27
3.2 ESPECIFICAÇÃO 28
3.2.1 APLICATIVO PEER 29
3.2.2 APLICATIVO SERVIDOR DE NOMES 35
3.3 IMPLEMENTAÇÃO 39
3.3.1 TÉCNICAS E FERRAMENTAS UTILIZADAS 40
3.3.2 PROCESSO DE IMPLEMENTAÇÃO 42
3.3.3 OPERACIONALIDADE DA IMPLEMENTAÇÃO 56
3.4 RESULTADOS E DISCUSSÃO 62
4 CONCLUSÕES 65
4.1 EXTENSÕES 66
REFERÊNCIAS BIBLIOGRÁFICAS 67

1 INTRODUÇÃO

Os limites impostos pelo desenvolvimento de tecnologias que possibilitem um aumento significativo no poder computacional de um processador doméstico, ou mesmo em supercomputadores, inviabilizam a resolução de inúmeros problemas computacionais atuais. Está-se presos a limites físicos (tamanho dos transistores, temperatura de funcionamento, etc) e financeiros, já que o poder de processamento custa caro e muitas vezes o custo/benefício não justifica o gasto.

Nesse contexto, muitas aplicações são tidas como inviáveis e são abandonadas, seja porque o projeto necessitaria de milhões de dólares para pagar o custo computacional, ou por conta do tempo que seria gasto com os recursos computacionais disponíveis. O usuário comum também passa por esse tipo de problema ao necessitar de um poder computacional não disponível em seu computador.

Observando-se o número de computadores interligados pela *internet* e analisando a porcentagem de quantos destes estão sempre rodando processos pesados enquanto navegam pela rede, descobre-se ociosidade de processamento na maioria dos casos, onde há computadores que estão sempre ligados, mas passam a maior parte do tempo sem um usuário logado nem processos rodando. O poder computacional desperdiçado nessas máquinas é enorme, sem falar da energia elétrica que está sendo gasta sem nenhum objetivo que não seja a máquina estar sempre pronta para um usuário se logar. Grandes e pequenas instituições que possuem computadores em rede utilizam esta postura de não desligar seus computadores no momento que o usuário os deixa.

Juntando essas informações (limites na tecnologia de processadores mais rápidos, aumento na largura de banda e no número de computadores interligados na *interne*t e a ociosidade que a maioria deles dispõe), foi que se originou a idéia de se utilizar a ociosidade

desses computadores para realizar tarefas colaborativas. O processamento distribuído funciona como um verdadeiro supercomputador virtual, no qual é possível configurar as tarefas de cada estação.

Para viabilizar a existência do processamento distribuído, existe um administrador que determina o que cada computador vai processar e quando esses dados já processados serão incorporados aos resultados do processamento. A heterogeneidade entre as máquinas participantes do processamento distribuído já mostra como problema, a capacidade, em termos de poder de processamento que cada máquina do processamento distribuído possui ser diferente, já que ele pode ser um computador comum ou um supercomputador. O administrador terá que tomar decisões na hora de incorporar esses resultados, sendo que o ideal é que haja mais independência possível de um resultado para o outro. O segundo problema se refere à plataforma de cada computador participante do processamento distribuído. Programas de código aberto são bem mais desejáveis para resolver este problema.

Segundo Yuri (2002), inúmeros projetos em processamento distribuído estão neste momento espalhados pelo mundo. O mais popular atualmente, e um dos primeiros a se difundir na rede é o *Seti@home* da Universidade de Berkeley. Para o projeto *Search for Extraterrestial Intelligence* (SETI, 2000), milhares de pessoas no mundo rodam um protetor de tela que analisa os sinais de rádio capturados pelo radiotelescópio. O *Seti@home* atualmente tem conseguido alcançar a marca de 1 zettaflop (um sextilhão de operações aritméticas por segundo). Na prática, o usuário que possui um computador, que não necessita estar constantemente ligado à *internet*, baixa um cliente do processamento distribuído da *Seti@home* compilado para sua plataforma e o instala. O usuário cria uma identificação e recebe neste momento um pequeno pacote contendo os dados que serão processados durante as horas ociosas do computador. Não é necessário, portanto estar conectado, o programa vai processando localmente e colecionando os resultados em um pacote de retorno. Concluída a

operação, quando o usuário volta a se conectar na rede, o programa envia o pacote resultado do processamento e solicita um outro pacote continuando a operação *off-line*. Os requisitos para o processamento distribuído deste tipo de aplicação são que, os pacotes de dados e resposta sejam pequenos para não ocupar a banda de dados dos clientes, e o processamento seja leve para não afetar as tarefas na máquina cliente.

Para pesquisar a existência de novos materiais na natureza, a Organização Européia de Pesquisas Nucleares (CERN, 1993), criou um consórcio mundial de laboratórios para montar um processamento distribuído para a execução de um acelerador linear.

Processamento distribuído pode ser utilizado para aproveitar o poder de processamento ocioso das máquinas pela rede. Por outro lado, problemas como a heterogeneidade das máquinas (plataforma e poder computacional), interdependência entre os dados a serem processados e a infra-estrutura que coleta todos esses dados gerando um resultado final, ainda necessitam de fortes estudos e realizações. Alguns trabalhos já buscam criar projetos de processamento distribuído genéricos onde um usuário possa escolher o que vai querer processar dentro do processamento distribuído e obter seu resultado de forma transparente.

Neste contexto, como proposta à utilização do processamento distribuído em um ambiente *peer-to*-peer, será necessária a implementação de um aplicativo servidor de nomes, responsável pelo gerenciamento e distribuição da lista de *peers* para os aplicativos *peers* conectados ao servidor de nomes e um aplicativo que será o *peer*, o qual se conecta ao servidor de nomes e, de posse da lista de *peers* solicita e disponibiliza processamento para outros *peers* da lista de *peers* distribuída pelo servidor de nomes.

Neste trabalho, o aplicativo *peer* será executado em alguns computadores de uma rede local e o servidor de nomes será executado em um computador também da rede local. Quando um *peer* for executado, ele acessará o servidor de nomes. Este *peer* torna-se conhecido para o servidor de nomes, sendo assim incluído na lista de *peers* que o servidor de nomes contém, o

servidor de nomes neste momento distribui a lista de *peers* a todos os aplicativos *peers* que estiverem na lista, podendo assim de qualquer *peer*, ser alocado o poder computacional ou ser atendido seu pedido de processamento. Quando for necessário executar um trabalho que requer bastante processamento por um *peer*, o *peer*, de posse da lista de *peers*, escala um ou mais *peers* para o processamento da tarefa. Esta troca de informações entre os computadores é o que caracteriza então a comunicação de *peer-to-peer*, ou seja, comunicação entre os aplicativos *peer* da rede de processamento, sem a necessidade de comunicação com o servidor.

Como serviços que cada *peer* disponibilizará ao ambiente, será implementado uma base de dados, contendo uma lista de itens que podem ser pesquisados por outro *peer*. Sendo assim, o serviço disponibilizado será o processamento utilizado na pesquisa de um determinado item na base de dados que o *peer* disponibilizou para a rede.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um protótipo de software que realize processamento distribuído em ambiente *peer-to-peer*, demonstrando assim a funcionalidade na resolução de problemas computacionais que necessitem de grande poder computacional.

Os objetivos específicos do trabalho são:

a) desenvolver um protótipo de um aplicativo servidor de nomes, que gerenciará uma lista de *peers*, e distribuirá esta lista de *peers* a todos computadores que, executando o protótipo do aplicativo *peer*, se conectarem ao servidor de nomes;

b) desenvolver um protótipo de um aplicativo *peer* que se conectará ao servidor de nomes e disponibilizará sua capacidade de processamento, bem como requisitará processamento para realização de suas tarefas;

c) utilizar vários computadores de uma rede local para formar o processamento distribuído, sendo que somente um computador atuará como servidor de nomes e os outros como *peers*. O computador definido como servidor de nomes poderá também executar o protótipo *peer*, ou seja, aplicativo servidor de nomes e *peer* executando no mesmo computador, evitando assim a ociosidade de um computador que execute somente o protótipo servidor de nomes.

1.2 ESTRUTURA DO TRABALHO

O presente trabalho está dividido em quatro capítulos, sendo que o primeiro compreende a introdução ao trabalho, com sua contextualização, seus objetivos e sua organização.

O segundo capítulo apresenta os conceitos, técnicas e ferramentas mais relevantes ao propósito do trabalho, bem como a descrição dos trabalhos correlatos.

O terceiro capítulo engloba a metodologia de desenvolvimento do trabalho, apresentando os requisitos principais do problema, a especificação, a implementação e a discussão dos resultados obtidos através de testes operacionais feitos a partir de um estudo de caso.

No quarto capítulo são apresentadas as conclusões obtidas com o desenvolvimento do trabalho, bem como suas vantagens, limitações e as sugestões de extensão para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo serão apresentados os principais conceitos a respeito das seguintes tecnologias: redes *peer-to-peer*; agentes móveis; *aglets* e XML. Também serão relatados os trabalhos correlatos.

2.1 REDES PEER-TO-PEER

De acordo com Rocha (2003, p. 1-2), a definição do que é uma rede do tipo *peer-to-peer* pode parecer obscura em um primeiro momento. Contudo, é clara a distinção entre o paradigma cliente-servidor e o paradigma *peer-to-peer*. No primeiro existe uma clara distinção entre a entidade que está provendo um serviço e o cliente que o está consumindo. Por outro lado, não existe distinção entre os elementos de uma rede *peer-to-peer*, todos os nós possuem funcionalidades equivalentes.

Embora a definição exata de *peer-to-peer* seja discutível, estes sistemas não possuem infra-estrutura centralizada, dedicada, mas ao invés disso, dependem da participação voluntária dos pares para contribuir com recursos para construção da infra-estrutura. A participação nos sistemas *peer-to-peer* é dinâmica.

Com o a aparecimento do sistema Napster (NAPSTER, 2000) este tipo de rede tornou-se extremamente popular. O Napster era utilizado para a troca de arquivos, geralmente música, entre os seus usuários, tornando-se muito utilizado e conhecido em pouco tempo. No ano de 2001 o Napster foi considerado a aplicação com maior crescimento na *web*, tendo sido feito *download* por cerca de 50 milhões de usuários. O mesmo foi desenvolvido por uma companhia privada de mesmo nome que foi processada pela indústria fonográfica devido ao fato de os usuários do sistema o utilizarem para a troca de músicas protegidas por direitos

autorais sem a devida permissão. Para utilizar o Napster, os usuários registravam-se no sistema identificando os arquivos disponibilizados pelos mesmos. Após isto, um outro usuário poderia consultar o servidor, ou *cluster* de servidores, para encontrar e transferir um arquivo anteriormente compartilhado por outro usuário.

Com o encerramento das operações do Napster, seus usuários e defensores começaram a utilizar outro sistema, o Gnutella (GNUTELLA, 2000). Diferentemente do seu predecessor, este sistema não necessita de uma entidade central para a realização das buscas aos arquivos, possibilitando a operação direta entre os usuários. Desta forma, a indústria da música viu-se impossibilitada de impedir o funcionamento do Gnutella, pois não se sabe quem são os indivíduos que estão trocando arquivos. Mesmo que se identifiquem alguns usuários que estejam infringindo a lei, a punição dos mesmos não irá impedir o funcionamento da rede e o restante dos usuários continuaria a permutar músicas.

Com isto fica claro que existem muitas motivações de teor não técnico para a utilização destes sistemas. Aproveitando assim o crescimento do poder de processamento dos computadores pessoais, que juntos somam uma capacidade extraordinária de processamento que não é utilizada nos dias de hoje. Segundo Shirky (1996), esta capacidade soma 10 bilhões de *megahertz* de poder de processamento e 10 mil *terabytes* de armazenamento, considerando que de 100 milhões de PCs dos 300 milhões de usuários da *internet*, cada um contribua com apenas 100 Mhz de processamento e 100 Mb de armazenamento.

Redes *peer-to-peer* para compartilhamento de arquivos na *internet* são resumidamente redes *overlay* dinâmicas com participação voluntária onde todos os nós participantes trocam arquivos diretamente. As mesmas podem possuir o mecanismo de busca centralizado ou distribuído pela rede. Para Clarke (1999), alguns sistemas *peer-to-peer* foram criados para expressão livre e anônima de idéias, como o *Freenet*. Neste, o objetivo não é trocar arquivos de música, mas expressar de forma livre as idéias e pensamentos políticos, sexuais ou de

qualquer natureza.

2.2 AGENTES MÓVEIS

Segundo Rubinstein (2001), um agente móvel (AM) é um agente de software que não se limita somente ao sistema em que iniciou sua execução, mas tem a habilidade de se autotransportar de um sistema para outro, juntamente com seu estado (estado de execução e variáveis) e continuar a execução de suas tarefas neste novo ambiente. Geralmente, são programados em linguagens interpretadas (como Java ou TCL) devido à necessidade de portabilidade.

No conceito de Ricarte (2000), um agente de software é um programa que age de forma autônoma em nome de uma pessoa ou organização, na maioria das vezes possuindo propriedades cognitivas, normalmente através de um procedimento interno de tomada de decisões alimentado por um conjunto de conhecimentos específicos. Resumidamente, um agente de software pode ser definido como um processo autônomo direcionado a metas.

Cada agente de software é executado como um processo distinto e desta forma suas tarefas são executadas por sua própria iniciativa e sem a necessidade de intervenção contínua do usuário (pró-atividade e autonomia) (MILOJICIC, 1998).

Para Aridor (1998), serviços de um agente de software são funcionalidades básicas relacionadas à migração e operação do AM, como:

a) movimentação: funcionalidade de transferência de um AM de um sistema para outro, que consiste em uma cópia do AM para o nó destino e após a confirmação de término desta operação, o AM é destruído no nó de origem;
b) criação: funcionalidade que permite a inicialização de um AM, que pode ser invocada pelo usuário da aplicação ou ainda por outro AM;

c) clonagem: permite que um AM possa ser copiado no mesmo local ou em outro nó da rede, sendo que este novo agente é igual em dados e estado de execução ao agente clonado, porém, receberá um novo identificador;

d) finalização: uma vez que sua tarefa tenha sido cumprida, o serviço de finalização permite que o sistema possa terminar a execução do AM;

e) retração: esta funcionalidade não aparece em todas as implementações de Sistemas de Agentes Móveis (MAS) ou *Mobile Agent Systems*, mas permite que solicitemos o retorno de um AM que já tenha migrado para outro local. Como exemplo do uso desta funcionalidade, pode-se citar a retração de AM por razões de segurança, permitindo que estes somente migrem para um determinado equipamento caso o mesmo o solicite;

f) persistência: permite a desativação temporária do AM, de forma que sua execução possa ser retomada exatamente na mesma situação em que o agente móvel foi desativado, sem nenhum prejuízo à execução de sua tarefa ou perda de seu estado e dados já colhidos ou processados. Neste processo, o estado e os dados são armazenados em memória não volátil;

g) comunicação: funcionalidade que permite a um AM ser capaz de localizar outros AM e trocar mensagens entre si – este serviço é realizado mais comumente através de um servidor de nomes compartilhado.

A utilização de um AM permite uma melhor utilização do tempo disponível do profissional responsável por realizar algum tipo de pesquisa, visto que o mesmo poderá disparar o agente e logo em seguida desconectar-se da rede para realizar outras tarefas, obtendo os resultados em outro momento quando a conexão for restabelecida.

2.2.1 AGLETS

Segundo Lange e Oshima (1998, p. xxii), *aglets* representam o próximo passo adiante na evolução do conteúdo executável na *internet*, introduzindo conceito de código de programa que pode ser transportado em estado de informação. *Aglets* são objetos Java que podem mover-se de um computador para outro, isto é, um *aglet* que executa em um computador pode de repente parar a execução e ir para um outro computador remoto, continuando então a execução neste.

Ainda citando Lange e Oshima (1998, p. xxii), *aglets* são agentes móveis em Java que suportam os conceitos de autonomia, execução e destinos dinâmicos no seu itinerário. Pode-se pensar sobre *aglets* como uma generalização e extensão de Java *applet* e *servlet*. *Aglets* são hospedados por um servidor de *aglets*, de uma maneira parecida como *applets* são hospedados por um navegador *web*. O servidor *aglet* fornece ambiente para *aglets* executarem e o *JVM (Java Virtual Machine)*, gerenciador de segurança dos *aglets*, recebe e hospeda com segurança.

Logo, *aglets* são objetos Java com a capacidade de se mover de uma máquina para outra em uma rede, levando consigo o código de programa e o estado dos objetos que compõe o *aglet.* A migração inicia com a interrupção da execução do *aglet* na máquina origem, seu despacho para uma máquina remota e o reinício da execução após sua chegada ao destino. Mecanismos de segurança impedem que *aglets* não autorizados (*untrusted*) tenham acesso a determinados recursos do sistema.

De acordo com Lange (1998a, p.12), um *aglet* passa a maior parte de seu ciclo de vida em um contexto de execução. Isto acontece porque, exceto pelos momentos em que é transferido, um *aglet* sempre possui um local de trabalho específico. Em outras palavras, no momento em que um *aglet* é criado, enquanto está cumprindo tarefas ou no instante em que é

destruído, pode-se dizer que um *aglet* está sempre associado a um contexto de execução. Até mesmo quando se movimenta em uma rede, um *aglet* sai de um contexto apenas para se dirigir a outro.

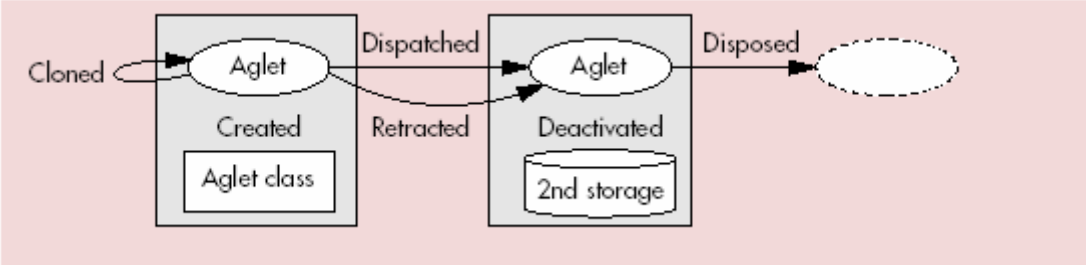
A interface particular de um *aglet*, que evita o acesso direto a seus métodos públicos, é denominada *proxy* (LANGE, 1998b, p.12). O *proxy* pode ser considerado então o representante de um *aglet*. Uma função importante do *proxy* é fornecer transparência de localização aos agentes, isto é, o *proxy* atua no sentido de ocultar a localização real de seu *aglet* correspondente. Numa situação de transferência de *aglet*, por exemplo, a comunicação com o agente continua sendo realizada da mesma maneira que antes do deslocamento (LANGE, 1998b, p.12).

A característica de autonomia é efetivamente garantida através de mecanismos como *threads*, que permitem a execução de agentes com algum grau de independência em relação aos outros. Também é característica necessária em um sistema de agentes a existência de um processo servidor que controle as configurações de acesso, segurança e comunicação do sistema. Além disso, a implementação de uma API para manipulação de agentes é uma diferença importante em relação aos objetos. Se a utilização de um sistema de objetos não requer nenhum servidor especial ou API, e não implementa na prática a característica de autonomia dos agentes, tal sistema deve ser considerado apenas como um sistema distribuído orientado a objetos. Uma vez que no sistema *aglets* estes três elementos - autonomia, servidor e API - estão presentes, considera-se que os *aglets* desenvolvidos nesta plataforma são autênticos agentes de *software*, em vez de simples objetos (LANGE, 1998, p.2).

A plataforma *aglets* disponibiliza um gerenciador que controla o tratamento de mensagens passadas concorrentemente. As mensagens passadas entre os *aglets* permitem interação flexível e troca de conhecimento entre os diversos sistemas (KARJOTH, 1997, p.2).

De acordo com Karjoth (1997, p.3), os agentes têm os seguintes estados em seu ciclo

de vida: criado, clonado, enviado, recolhido, desativado e disposto. Conforme apresentado na figura 1, pode-se visualizar que o *aglet* após ter sido criado (*created*), pode ser enviado (*dispatched*) a outro ambiente, ou ser clonado (*cloned*) e ao ser transferido, pode ser chamado de volta através do comando (*retracted*), desativado (*deactivated*) ou terminado (*disposed*).



Fonte: Karjoth (1997 p. 3)

Figura 1 - Ciclo de vida de um *aglet*.

2.3 XML (EXTENSIBLE MARKUP LANGUAGE)

O padrão XML foi inventado a partir da SGML (*Standard Generalized Markup Language*), linguagem de marcação padrão generalizada usada por muitos anos. A própria HTML é um aplicativo SGML, e o XML é um subconjunto da SGML ideal para ser usada na *Internet*. Então podemos dizer que todo documento XML é um documento SGML, e nem todo documento SGML é um documento XML. O XML simplifica a SGML já que remove todas as opções que não são absolutamente necessárias na SGML (LOTAR, 2001 p. 3).

Segundo Martinsson (2002 p. 21), os analisadores sintáticos SGML são capazes de extrair conteúdo e apresentar documentos SGML. O SGML é uma linguagem de marcação generalizada, que é usada como base para criar linguagens de marcação específicas e foi isso que levou a linguagens de marcação moderna.

O XML é um padrão de armazenamento de dados em um formato de texto simples, o que significa que o mesmo também pode ser aberto em qualquer computador. Um documento XML é formado por várias partes diferentes. As partes mais importantes são os elementos

XML que contém os dados reais para o documento.

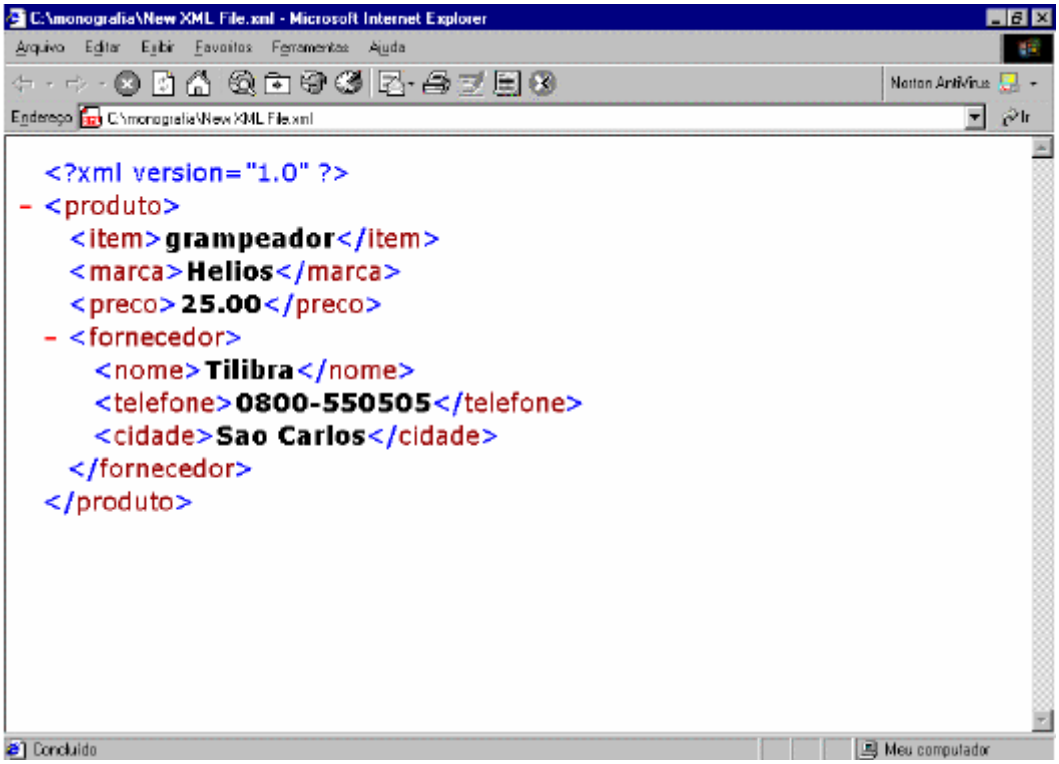
Para Furtado Júnior (2003), o XML é uma representação estruturada dos dados, que mostrou ser amplamente implementável e fácil de ser desenvolvida. Implementações industriais na linguagem SGML mostraram a qualidade intrínseca e a força industrial do formato estruturado em árvore dos documentos XML.

De acordo com Martinsson (2002 p. 22), o XML tornou-se uma recomendação da W3C (*World Wide Web Consortium*). Os documentos XML baseiam-se em texto, portanto, um documento XML pode ser transportado e compreendido entre máquinas interconectadas.

Ainda citando Martinsson (2002 p. 23), o XML complementa o HTML (*Hipertext Markup Language*), em vez de substituí-lo. Imagina-se o XML como um mecanismo que separa conteúdo e estrutura das informações que são apresentadas. O XML é uma solução mais para documentos sofisticados do que para documentos HTML.

Conforme Lotar (2001 p.7), quando usamos documentos XML realmente notamos que algo diferente acontece, já que temos todas as informações identificadas, além disso, o programador sente que tem o controle sobre o que está desenvolvendo, pois as tecnologias auxiliares a XML são fáceis de utilizar e também muito eficientes para manipular todas as informações existentes em um documento.

Apesar de ser simples e de possibilitar que se invente a própria linguagem de marcação e criação de novos elementos, é preciso se seguir regras definidas em um padrão que é bastante simples, mas suficiente providenciando que seja garantida à estruturação adequada da informação. Conforme pode ser visto no quadro 1, onde tem-se um exemplo de um arquivo XML, o nome do produto e seus atributos são facilmente identificáveis.



Quadro 1 - Exemplo de um documento XML

2.4 TRABALHOS CORRELATOS

No trabalho de conclusão de curso concluído por José Voss Junior (VOSS JUNIOR, 2004), tendo como orientador o Prof. Francisco Adell Péricas no 1º semestre de 2004, desenvolveu-se um protótipo de software em redes P2P para compartilhar arquivos entre os computadores envolvidos na rede, baseado inteiramente no projeto JXTA e usando a linguagem de programação Java. Apesar de não ser usado na implementação do protótipo o conceito de processamento distribuído, considera-se semelhante ao trabalho aqui proposto, pela utilização em ambos da tecnologia *peer-to-peer*.

Podendo citar ainda como trabalho correlato o trabalho concluído por Cristiano Marcio Borchardt (BORCHARDT, 2002), que consistiu na pesquisa sobre o funcionamento de sistemas de processamento de transações em ambientes distribuídos e as técnicas para garantir o funcionamento das propriedades ACID de transações. Como resultado obteve-se a

implementação de um mecanismo de gerenciamento de transações distribuídas através do protocolo de consolidação de duas fases e do algoritmo de registro de *log* de “refazer”, juntamente com uma classe de gravação de arquivos para serem utilizados por desenvolvedores que utilizam a ferramenta Visual C++, realizando a comunicação através da especificação de objetos distribuídos DCOM da Microsoft. Tendo como orientador o Prof. Marcel Hugo no 2º semestre de 2002.

No trabalho desenvolvido por Fernando Liesenberg (LIESENBERG, 2004), apresenta-se uma utilização do paradigma de agentes móveis. Apresenta também a especificação e implementação de um protótipo de software que utiliza agentes móveis para fazer monitoramento de computadores em uma rede. Para a implementação dos agentes móveis foi utilizada a linguagem Java. A linguagem C também foi utilizada para implementar uma DLL, para obtenção de informações sobre recursos do sistema operacional Windows. A DLL é acessada através do agente implementado em Java. Os mesmos agentes móveis foram implementados nas plataformas SACI e Aglets. A compatibilidade entre as plataformas é discutida. Tendo como orientador o Prof. José Roque Voltolini da Silva.

O Relatório final do projeto de pesquisa PIBIC/CNPq, desenvolvido por Alexandre Rodrigues Coelho (COELHO, 1998), apresenta os resultados obtidos no projeto que teve por objetivo principal especificar uma linguagem de descrição de protocolos de comunicação entre agentes. Chegando a conclusão de que tanto o paradigma de programação orientado a agentes distribuídos quanto as linguagens de comunicação entre eles estão suficientemente desenvolvidos para utilização na construção de sistemas onde os princípios de IAD possam ser aplicados.Tendo como coordenador do projeto de pesquisa o Prof. Jomi Fred Hübner.

3 DESENVOLVIMENTO DO PROTÓTIPO

Neste capítulo serão apresentados os requisitos do trabalho, bem como a sua especificação e ferramentas utilizadas para a implementação, o teste de operacionalidade e a discussão dos resultados obtidos.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

O protótipo proposto neste documento tem como requisitos funcionais (RF):

a) servidor: o protótipo deverá possuir um programa servidor que será responsável pela manutenção da lista de *peers* conectados a ele e pela distribuição desta lista de *peers* aos *peers* que a ele se conectar;
b) *peer*: o protótipo deverá possuir um programa *peer* que fará uma leitura dos arquivos XML que estão em um determinada pasta do sistema, os quais disponibilizará aos outros *peers* para consulta, ainda receberá a lista de *peers* do servidor de nomes e de posse desta lista de *peers* fará consultas aos arquivos XML que os *peers* da lista de *peers* possuírem;
c) tarefas: o aplicativo *peer* fará consultas a outros *peers* da lista de *peers* nos arquivos XML que estes possuírem;
d) resposta: o programa *peer* deverá informar ao *peer* que solicitou a tarefa, a resposta à tarefa que lhe foi enviado para processamento, ou seja, após o término da pesquisa do arquivo XML, deve-se enviar o resultado da pesquisa ao *peer* que solicitou.

Os requisitos não funcionais (RNF) resumem-se a:

a) arquitetura: o protótipo deverá possuir a arquitetura *Peer-to-Peer*;

b) comunicação: o protótipo deverá utilizar o conceito de processamento distribuído;

c) plataforma: o protótipo deverá ser compatível com o sistema operacional *Microsoft Windows 2000 Professional* ou superior;

3.2 ESPECIFICAÇÃO

O método de ciclo de vida escolhido para o desenvolvimento do protótipo apresentado neste trabalho é o iterativo e incremental, por se tratar de um método onde todo o trabalho é dividido em partes menores que tem como resultado um incremento ao trabalho final. Este método oferece maior segurança no atendimento dos requisitos e maior flexibilidade durante todo o processo de desenvolvimento.

A especificação do problema apresenta-se através de um grupo de diagramas da linguagem *Unified Modeling Language* (UML), sendo estes: diagrama de classes, diagrama de seqüência e diagrama de caso de uso. Utilizou-se a ferramenta *Jude Community* para a modelagem da especificação. Uma ferramenta que teve resultado satisfatório às necessidades das especificações necessárias ao desenvolvimento dos diagramas utilizados no trabalho.

Considerando o fato de que o problema trata de dois protótipos distintos, *peer* e servidor de nomes, apresentam-se o conjunto de diagramas para cada aplicativo separadamente.

3.2.1 APLICATIVO PEER

O diagrama de casos de uso do aplicativo *peer*, mostrado na figura 2, apresenta os casos de uso pertinente ao usuário do protótipo do aplicativo *peer*, que são: inicializaPeer, criar consulta, executar busca, visualiza status, visualiza bases e visualiza resultados.

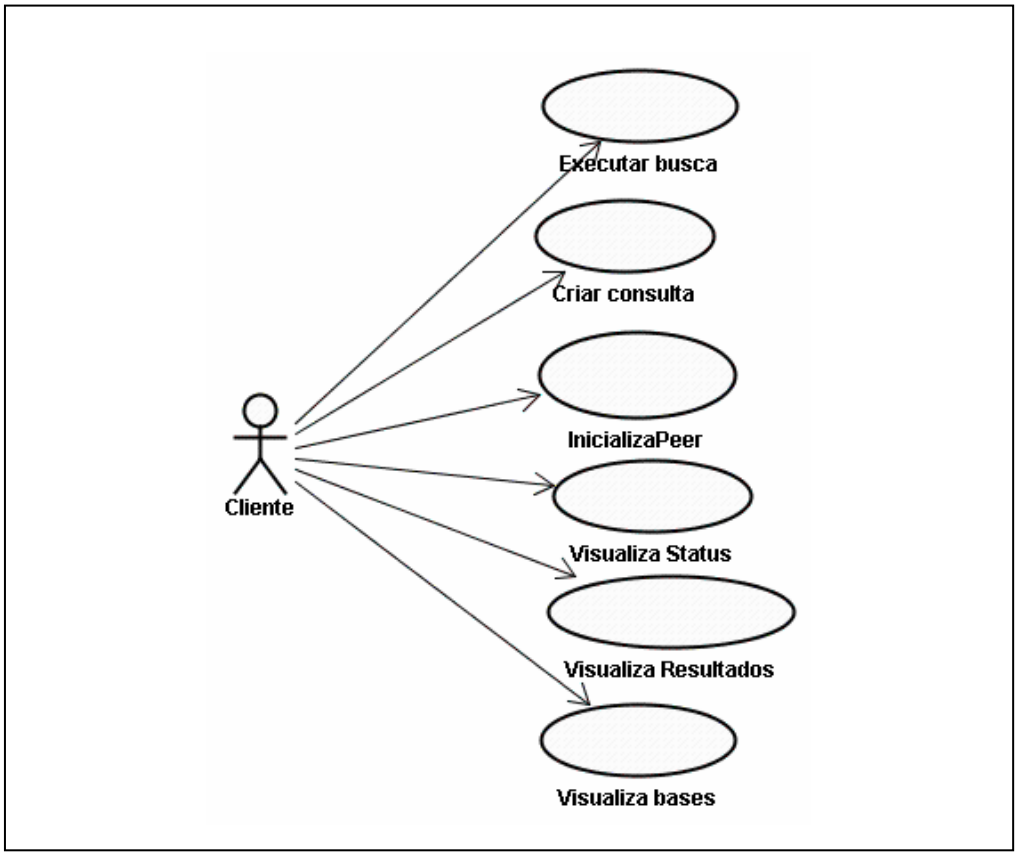


Figura 2 - Diagrama de caso de uso do aplicativo Peer.

O caso de uso “criar consulta” refere-se à funcionalidade que permite ao usuário configurar uma nova busca, definindo qual base de dados será pesquisada e quais os atributos desta consulta, de acordo com os atributos disponíveis da base selecionada para consultar. Enquanto que o caso de uso “executar busca” refere-se à execução da pesquisa definida, disparando a pesquisa nos *peers* que contiverem a base escolhida no caso de uso “criar consulta”.

Já o caso de uso “visualiza status” diz respeito a toda mensagem que é mostrada na barra de status do protótipo, sendo que a mensagem pode ser referente ao protótipo *peer* como

também pode trazer mensagens referentes ao status do servidor de nome. Há o caso de uso "visualiza bases", que mostra as bases de dados disponíveis a serem pesquisadas, sendo que estas bases podem estar em qualquer dos protótipos instanciados. Há também o caso de uso "inicializa *peer*" que é responsável por executar o aplicativo *peer* e ainda há o caso de uso "visualiza resultados", que mostra ao usuário o resultado da pesquisa.

A figura 3 apresenta o diagrama de classes do protótipo *peer*, sendo estas as seguintes: *AgletPeer*, *TelaPeer*, *AgletConsulta*, *TelaConsulta*, *TelaResultado* e *Agletmsg*. A classe *AgletPeer* é responsável por todos os processos que envolvem as classes *Agletmsg* e *AgletConsulta*, como: instanciação das classes, troca de mensagens e controle das mensagens vindas do servidor de nomes. Enquanto que a classe *TelaPeer* é responsável pela aparência gráfica do *peer* e dos dados e mensagens enviados pela classe *peer*.

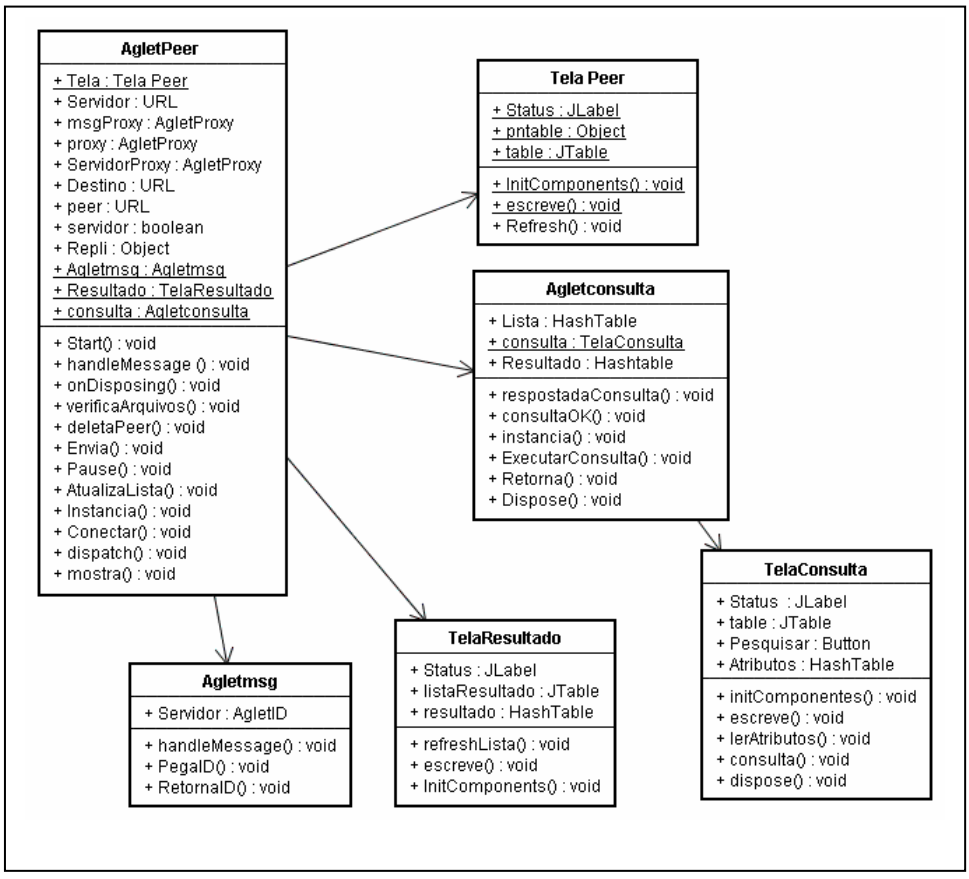


Figura 3 - Diagrama de classes do protótipo do aplicativo *peer*.

A classe *AgletConsulta* é responsável por todos processos que envolvem a consulta, como: instanciação da classe *TelaConsulta* e também pela pesquisa nos arquivos XML e ao

envio do *agletConsulta* à todos os *peers* que tenham a base à ser pesquisada. Para a classe *TelaConsulta* fica a responsabilidade de apresentar graficamente os controles da pesquisa que tornam-se visíveis no momento que é criada uma nova consulta.

A classe *TelaResultado* apresenta graficamente os resultados obtidos com a consulta aos arquivos XML dos *peers*, resultados estes recebidos através da classe *Peer*. Fica a encargo da classe *Agletmsg* a função de se mover até o ambiente onde se encontra o servidor de nomes, pegar o número de identificação (*AgletID*) do servidor de nomes e enviar à classe *Peer*, para esta poder então pegar seu *proxy*. *Proxy* é uma classe com todos os dados referentes aos *aglets*, necessários para haver troca de mensagens entre os *aglets*.

Quanto aos diagramas de seqüência, a figura 4 apresenta o caso de uso *InicializaPeer*, onde o objeto da classe *AgletPeer* aciona primeiramente o método *start()*, o qual irá instanciar um objeto do tipo *TelaPeer*. Que fica responsável pela parte gráfica deste *AgletPeer*.

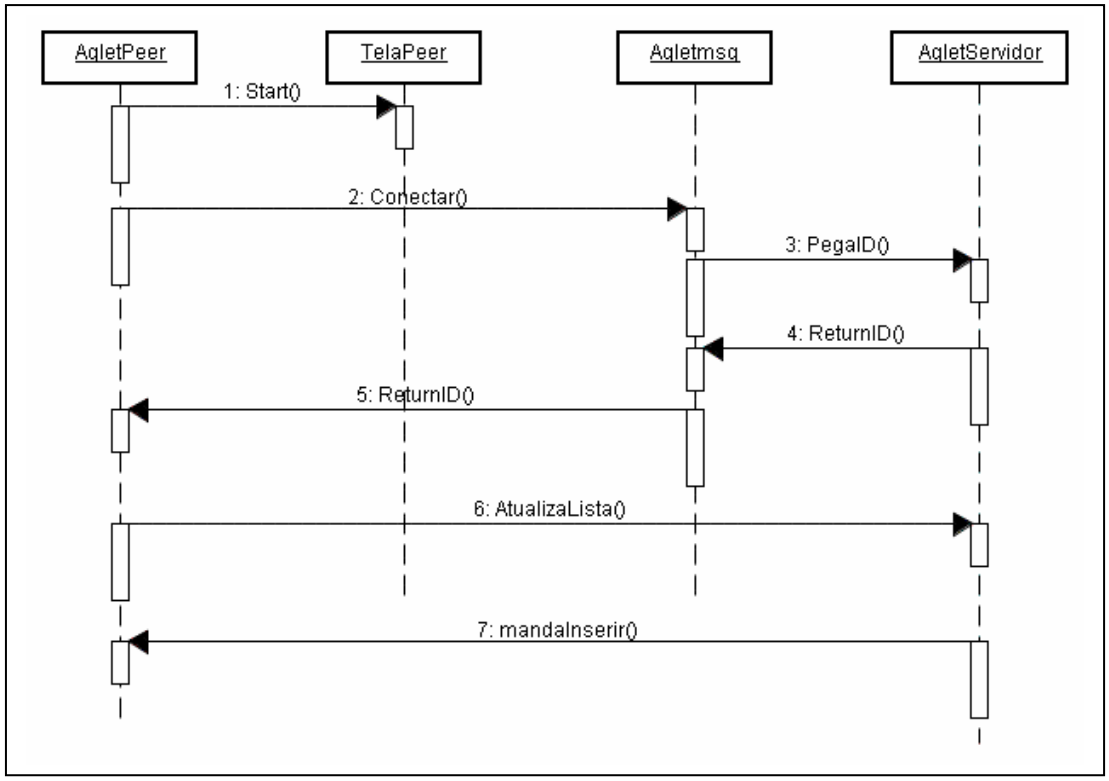


Figura 4 - Diagrama de seqüência do caso de uso *InicializaPeer* do *AgletPeer*.

Depois é executado o método *conectar()*, o qual é responsável por instanciar um objeto

do tipo *Agletmsg* e despachá-lo para o ambiente do servidor de nomes, este *Agletmsg* é responsável por enviar uma mensagem para o servidor de nomes para que ele passe ao *Agletmsg* seu id, com o método *pegaID()*, com posse do número de identificação (id) do servidor de nomes, o *Agletmsg* executa o método *ReturnID()*, o qual retorna o id do servidor de nomes ao *Agletpeer*, este *aglet*, por sua vez executa o método *AtualizaLista()*, que manda uma mensagem ao servidor de nomes para atualizar a lista de *peer* conectados ao servidor de nomes, então envia uma mensagem com a lista de *peer* conectados, para todos os peers conectados ao servidor de nomes através do método *mandaInserir()*.

A figura 5 ilustra o diagrama de seqüência do caso de uso *CriarConsulta()*, sendo que o *AgletPeer* executa o método *instancia()*, o qual é responsável pela instanciação do objeto *AgletConsulta* e este por sua vez dispara o método *instancia()*, que cria um novo objeto do tipo *TelaConsulta* que apresenta de forma gráfica todos os possíveis atributos da consulta.

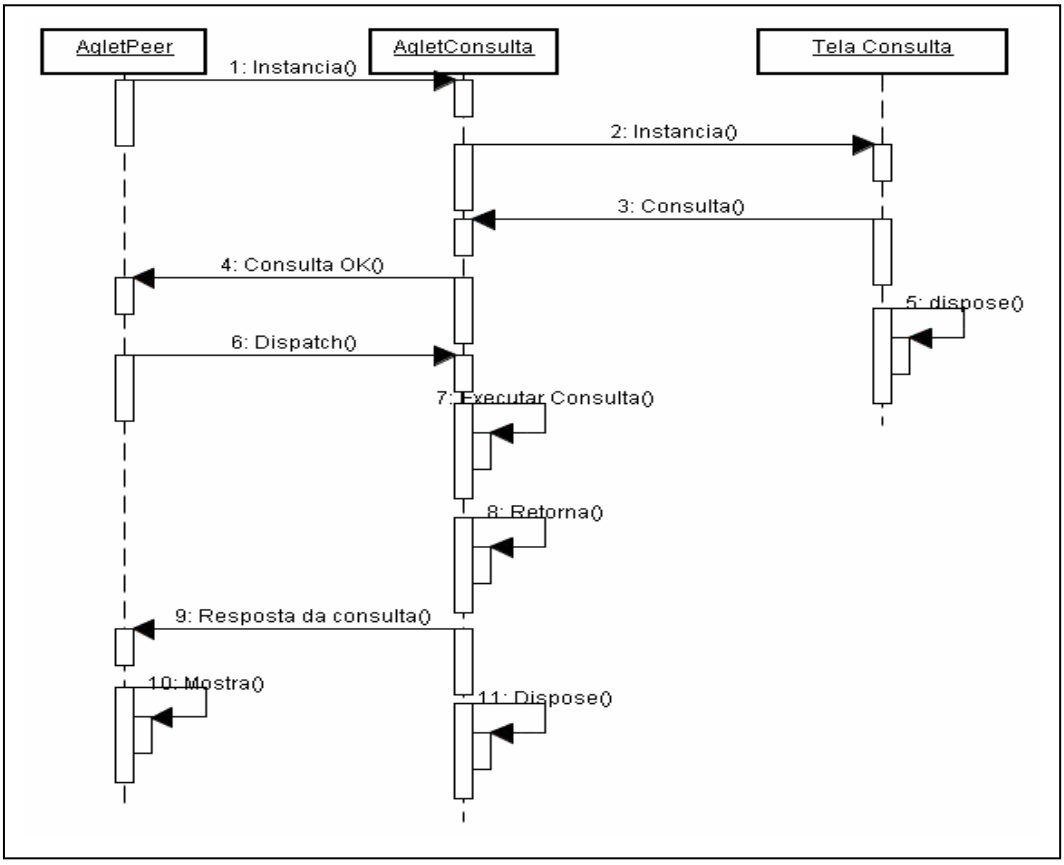


Figura 5 - Diagrama de seqüência dos casos de uso *CriarConsulta*.

Quando o usuário determinar os atributos da consulta, o usuário pressiona o botão

pesquisar e chama o método *Consulta(),* através do clique do botão enviar, que retorna ao *AgletConsulta* a forma como foi especificada a consulta, este então ativa o método *ConsultaOK(),* que envia uma mensagem ao *AgletPeer* notificando que a consulta foi elaborada.

Após ter enviado os atributos da consulta ao *AgletConsulta,* o *TelaConsulta* chama o método *Dispose()* que termina com o objeto *TelaConsulta*, quando o usúario pressionar o botão Pesquisar da TelaConsulta o *AgletPeer* dispara o método *Dispatch()* que clona o *AgletConsulta* e envia para todos os *peers* da lista de *peers* disponibilizada pelo servidor de nomes, que possuírem a base de dados que esta sendo elaborada a consulta, por exemplo, sendo a consulta elaborada para a base de dados carros, será enviado uma cópia do *AgletConsulta* para todos os *peers* que possuírem a base de dados carros.

Após terem sido enviados os clones do *AgletConsulta* para os *peers*, será processado o arquivo XML que contenha o nome da base de dados que foi elaborada a consulta. Assim, se a consulta foi elaborada para a base de dados carros, o arquivo XML a ser processado terá o nome de carros. Desta forma após ter sido processado o arquivo, o aplicativo *peer* através do método *retorna(),* envia uma mensagem ao *peer* que solicitou o processamento e executa o método *Dispose(),* que mata o clone do *AgletConsulta.* Com o resultado da busca de todos os clones o *AgletPeer* chama o método *mostra()*, que mostrará o resultado da consulta através do caso de uso *MostraResultado*.

Na figura 6 é representado o diagrama de seqüência do caso de uso *MostraResultado*, onde o *AgletPeer*, após ter recebido o resultados da pesquisa de algum dos *agletConsulta*, chama o método *instancia()*, que criará um objeto do tipo *TelaResultado* e em seguida o objeto *TelaResultado* ativará o método *RefreshLista(),* este atualizará a tabela dos resultados obtidos pela consulta, para que o objeto do tipo *TelaResultado* mostre-os de forma gráfica. O objeto *TelaResultado* ficará a disposição do usuário para conferencia dos resultados, até

quando ele decidir terminar o objeto com o método *close()*.

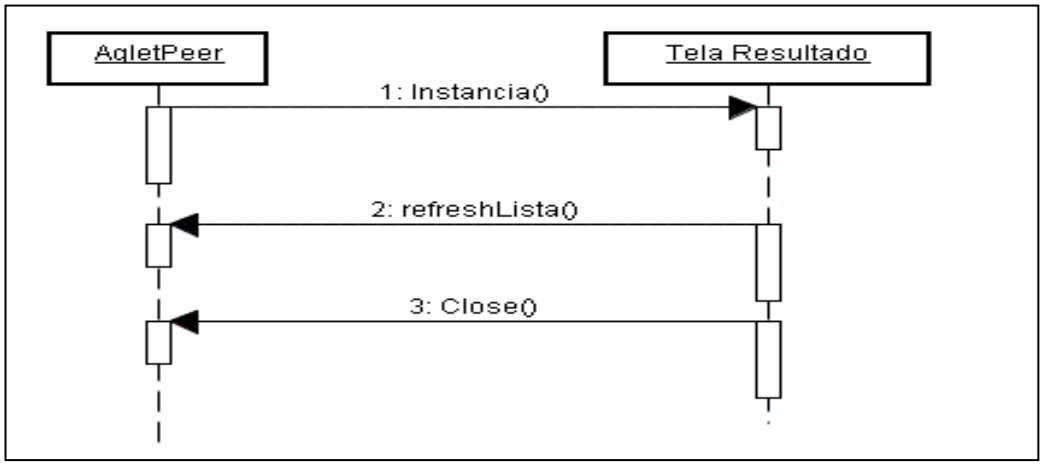


Figura 6 - diagrama de seqüência do caso de uso *VisualizaResultado*.

A figura 7 representa o diagrama de seqüência do caso de uso, nomeado de *VisualizaBasesDisponiveis()*, onde o *AgletPeer* ativa o método *VerificaArquivos()* que examina um diretório padrão em busca de arquivos XML, identifica os arquivos e executa o método *Refresh()* do objeto *Peer*, este fica encarregado de atualizar a lista de arquivos disponíveis e mostrar ao usuário as bases de pesquisa disponíveis. O próximo passo é o chamado do método *Dispatch()* do objeto *AgletPeer* que enviará o objeto do tipo *AgletConsulta* aos *peers* registrados no servidor de nomes.

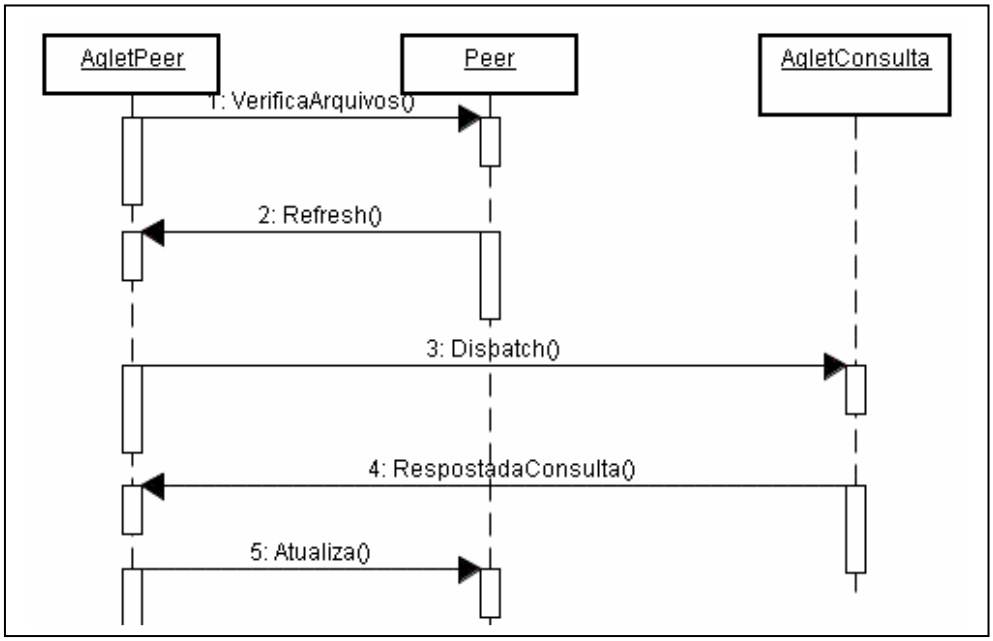


Figura 7 - diagrama de seqüência do caso de uso *VisualizaBasesDisponiveis*.

Após o retorno de todos os *AgletConsulta* com a resposta da consulta ao *AgletPeer*

com o método *RespostadaConsulta()*, o objeto do tipo *AgletPeer* executa o método *Atualiza()*, que envia a resposta da consulta para ser mostrada ao usuário e atualiza a mensagem da barra de *status* do objeto *peer*, através do método *Refresh()*.

3.2.2 APLICATIVO SERVIDOR DE NOMES

A especificação do aplicativo servidor de nomes é apresentada inicialmente pelo diagrama de casos de uso da figura 8, que ilustra os seguintes casos de uso: VisualizaPeer, InserePeer, ExcluiPeer e onDisposing.

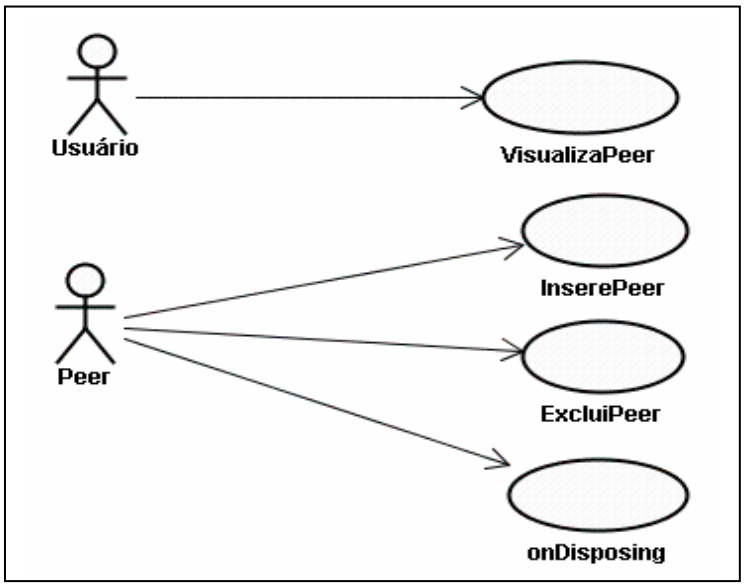


Figura 8 – Diagrama de casos de uso do aplicativo Servidor de nomes.

Sendo que o usuário poderá executar somente uma funcionalidade que é *visualizaPeer()*, que refere-se ao procedimento de dar um clique sobre um peer da lista de *peer* da tela do servidor para visualizar todos os dados referentes ao objeto *peer*. As demais funcionalidades são executados pelo próprio *peer*, que são *InserePeer()*, responsável por todos os procedimentos envolvidos na inserção de um *peer* na lista de *peers* conectados ao servidor de nomes.

O caso de uso *ExcluiPeer()* é composto pelos métodos envolvidos com a exclusão de

determinado *peer* da lista de *peer*, e por fim o caso de uso *onDisposing()* faz referencia aos métodos relacionados com a finalização do aplicativo servidor de nomes.

O aplicativo servidor de nomes, conforme apresenta a figura 7, é composto pelas seguintes classes: *AgletServidor* e *Servidor*.

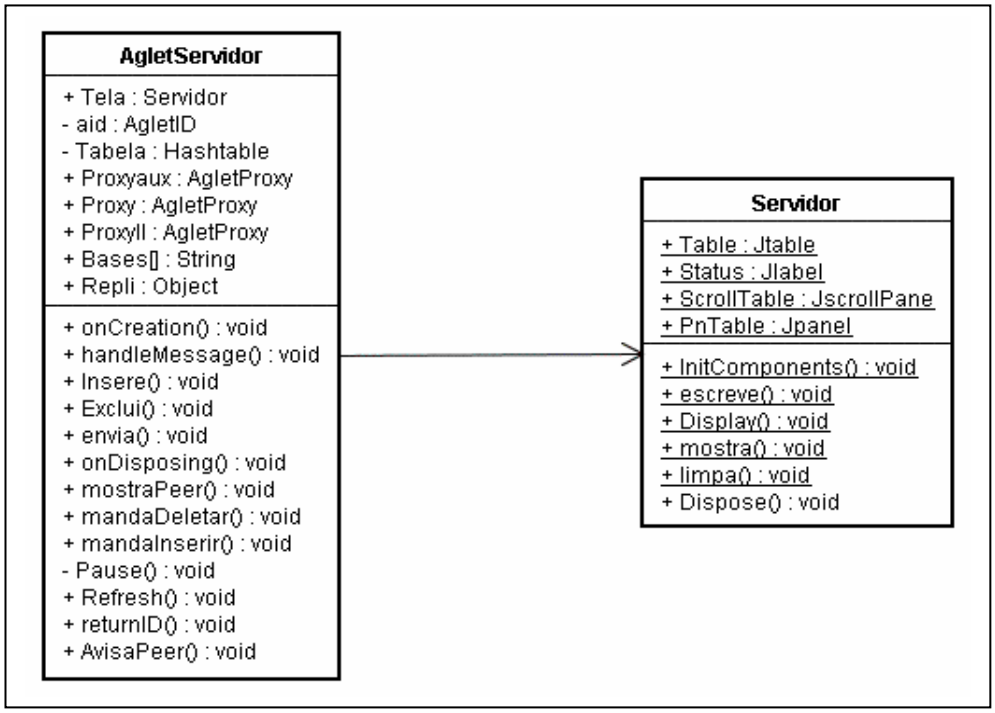


Figura 9 – Diagrama de classes do aplicativo servidor de nomes.

A classe *Servidor* é responsável pela aparência gráfica da aplicação servidor de nomes, composto por uma tabela onde ficarão armazenados de forma visível ao usuário a lista de *peers* conectados ao servidor de nomes. Para a classe *AgletServidor*, fica a responsabilidade de gerenciar os *peers* que a ele se conectarem, pois um *AgletPeer* ao ser inicializado tenta se conectar ao servidor de nomes, a fim de manter a lista de *peers* conectados atualizada e para que o servidor de nomes envie para este *peer* a lista de *peer* conectados a ele, para que a partir deste momento o *peer* possa trocar mensagens com os outros *peers* que estão conectados ao servidor de nomes.

Quanto aos diagramas de seqüência do servidor de nomes, a figura 10 apresenta o caso de uso visualizaPeer, onde o objeto da classe *Servidor* aciona o método *mostra()*,que captura

o clique do mouse e passa ao *AgletServidor* o ID do *peer* escolhido. Em seguida o *AgletServidor* chama o método *mostraPeer()*, que de posse do ID do *peer* pega o *proxy* deste *peer* e passa todas as informações referentes ao *peer* para o objeto da classe *Servidor*, este então exibe as mensagens do *peer* escolhido.

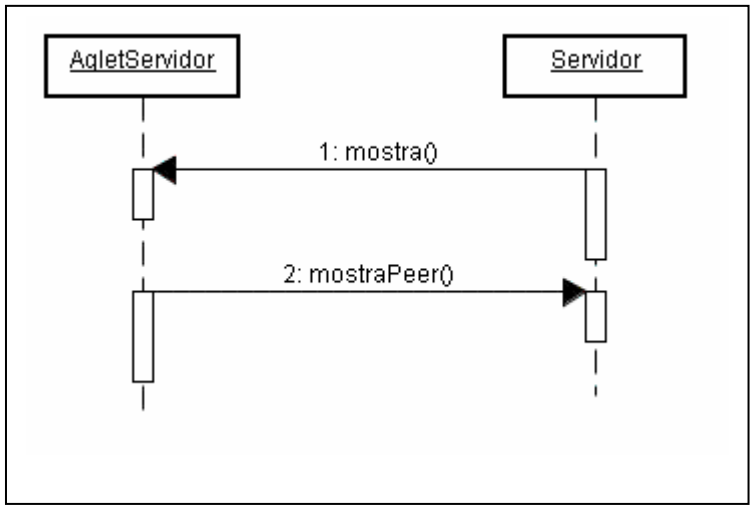


Figura 10 – Diagrama de seqüência do caso de uso *VisualizaPeer*.

A figura 11 ilustra o diagrama de seqüência do caso de uso *inserePeer()*, que ocorre na inicialização de um novo aplicativo *peer*.

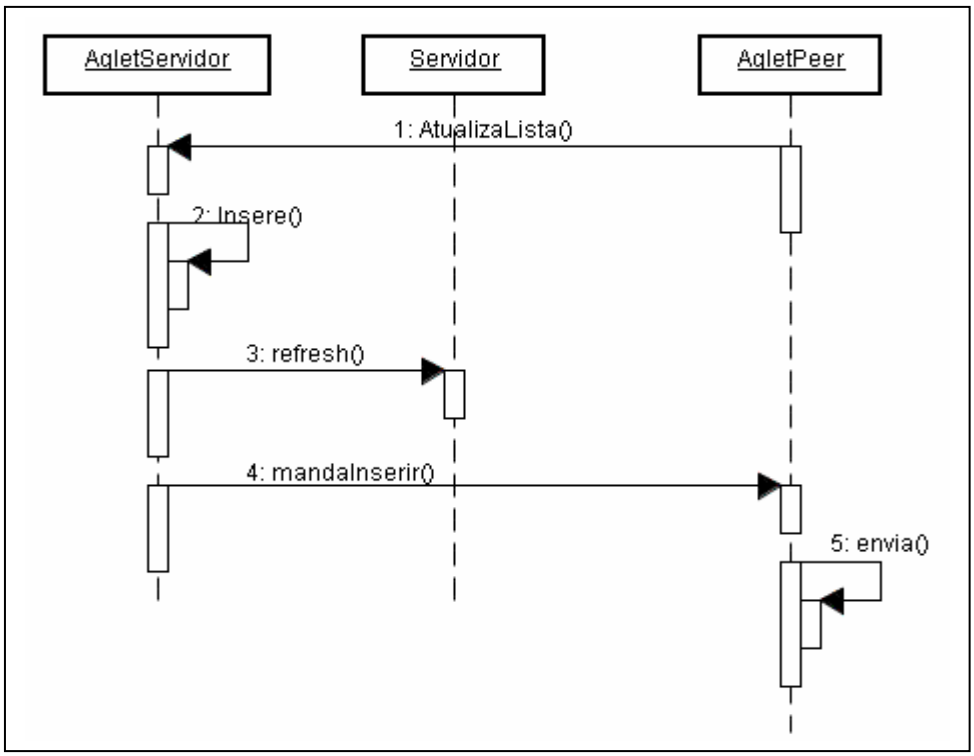


Figura 11 – Diagrama de seqüência do caso de uso *InserePeer*.

O objeto da classe *AgletPeer* chama o método *AtualizaLista()*, o qual envia uma mensagem para o objeto da classe *AgletServidor* com seu *proxy*, este por sua vez aciona o método *insere()*, que é responsável pela inserção do *proxy* deste novo *peer* na lista de *peers* conectados ao servidor de nomes.

Em seguida o objeto da classe *AgletServidor* executa o método *refresh()*, responsável por enviar a nova lista de peers para atualizar a tabela do objeto da classe *Servidor*. Após o objeto da classe *Servidor* ter atualizado sua lista é a vez dos *peers* que estão conectados ao servidor receberem a nova lista de *peer*, através da chamada do método *mandaInserir()* pelo *AgletServidor*, depois o *AgletPeer* invoca o método *envia()*, que é a atualização da linha de status do objeto *TelaPeer*, neste caso da utilização do método *envia()*, a linha de status receberá a quantidade de *peers* que estão conectados ao servidor de nomes.

Na figura 12 está representado o diagrama de seqüência do caso de uso *ExcluiPeer*, que ocorre na finalização de um objeto da classe *AgletPeer*. O objeto da classe *AgletPeer* executa o método *onDisposing()* que é responsável por, na finalização do objeto, enviar ao servidor de nomes uma mensagem para excluir o *peer* da lista de *peers* conectados.

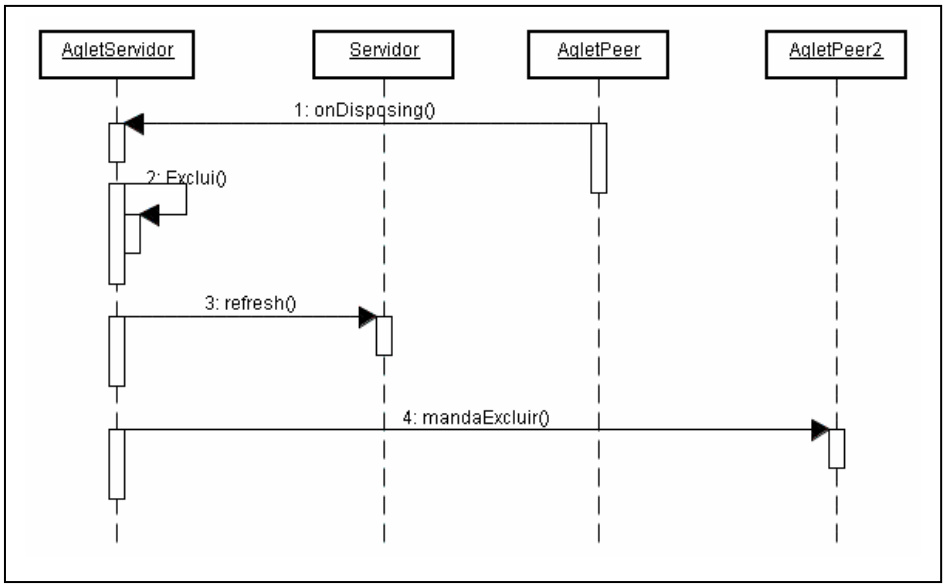


Figura 12 – Diagrama de seqüência do caso de uso *excluiPeer*.

O objeto da classe *AgletServidor* chama o método *Exclui()*, encarregado de excluir o

peer da lista de *peers* conectados e em seguida ativa o método *refresh()*, que tem por finalidade a atualização da tabela de *peers* conectados ao objetos *Servidor*, depois deste passo, o objeto da classe *AgletServidor* ativa o método *mandaExclui()*, que envia uma mensagem aos outros objetos da classe *AgletPeer* que estão conectados ao servidor de nomes, para que estes possam excluir este *peer* que foi finalizado, a fim de manter sua lista de *peers* também atualizada.

O diagrama de seqüência do caso de uso *onDisposing* é apresentado na figura 13, onde o servidor de nomes, ao ser terminado, chama o método *avisaPeer()*, responsável por enviar uma mensagem caracterizando a finalização do servidor de nomes, a todos objetos da classe *AgletPeer* que estiverem contidos em sua lista de *peers*.

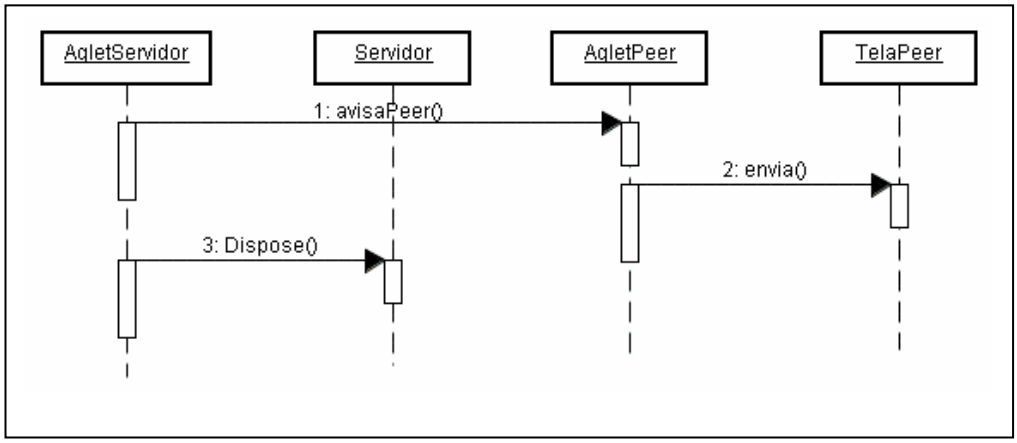


Figura 13 – Diagrama de seqüência do caso de uso *onDisposing*.

Os objetos da classe *AgletPeer*, por sua vez ativam o método *envia()*, que atualiza a barra de status do objeto *TelaPeer* com uma mensagem de aviso sobre a finalização do servidor. Após isso o objeto da classe *AgletServidor* chama o método *dispose()*, responsável por finalizar tanto o objeto do tipo *Servidor* quanto o objeto do tipo *AgletServidor*.

3.3 IMPLEMENTAÇÃO

Esta seção apresenta as técnicas e ferramentas utilizadas no desenvolvimento do

trabalho como: método de desenvolvimento e implementação, ambiente e linguagem de programação.

São apresentadas também a explanação, com partes do código fonte, das funcionalidades mais relevantes do sistema, como:

a) a instanciação de um agente móvel;
b) a forma de identificação do mesmo;
c) como são transferidos os agentes móveis de um contexto para outro;
d) a comunicação síncrona e assíncrona entre os agentes, a comunicação de um agente direcionada a outro agente móvel ou a comunicação de um agente móvel a todos agentes móveis de um contexto na forma de mensagem *multicast*;
e) o registro desta mensagem *multicast* no agente que poderá receber esta mensagem;
f) os eventos *onDisposing* e *handleMessage*, de um agente móvel;
g) a forma de criação de uma mensagem simples ou a mensagem composta pela mensagem anexada ainda algum objeto;
h) a forma como pode-se clonar objetos do tipo agentes móvel;
i) a forma como um agente enviado a um determinado contexto retorna ao contexto original;
j) a visualização das informações relativas ao agente e a utilização do protocolo de transferência dos agentes (ATP).

3.3.1 TÉCNICAS E FERRAMENTAS UTILIZADAS

O processo de implementação deu-se ao término da especificação de cada nova parte do protótipo, seguido por uma fase de testes do tipo caixa aberta para verificar a funcionalidade da parte de implementada. E por fim, uma fase de testes tipo caixa fechada,

para comprovar a funcionalidade da nova parte com o restante da aplicação, bem como averiguar se a nova parte implementada não afetaria nenhuma outra parte já existente na aplicação.

A linguagem de programação escolhida para o desenvolvimento do trabalho apresentado foi a linguagem Java. Verificou-se durante a revisão bibliográfica a necessidade de utilizar um ambiente que possibilite o emprego do conceito de agentes móveis, para isto foi utilizada o *Aglets Software development Kit* ou simplesmente ASDK, uma plataforma para desenvolvimento de agentes móveis, denominado *Aglets*, Atualmente existem duas versões para a plataforma ASDK. A primeira, *Aglets Worbench* foi desenvolvida pelos pesquisadores *Danny Lange* e *Mitsuru Oshima*, da IBM Japão, no ano de 1996, porém esta versão da IBM ainda não é compatível com as versões mais recentes da linguagem Java, de forma que para o JDK 1.2 e superiores, uma versão de código-livre do *Aglets Software Development Kit*, o ASDK 2.0, do *SourceForge,* foi utilizada para a implementação do conceito de agentes móveis.

Os *Aglets* são executados em um servidor de agentes denominado *Aglet Server*. O *Aglet Server* fornece aos usuários um aplicativo gráfico, denominado *Tahiti*, que possibilita a criação, transferência e destruição dos agentes, além da configuração de privilégios de acesso e segurança no servidor. Trata-se de um visualizador do contexto de execução dos *aglets*, ou ainda, uma interface gráfica que gerencia os serviços oferecidos pelo *Aglet Server*. Porém para o presente trabalho foi usada somente a propriedade, privilégios de acesso e segurança no servidor.

Quanto ao ambiente de desenvolvimento, foi escolhido o *Eclipse* 3.0 para a plataforma *Windows*, por ser uma ferramenta robusta que tem possibilidade de ser incrementada com *plugins* e também pelo fato da ferramenta ter sido utilizado em trabalhos passados, criando assim um certo domínio sobre a ferramenta.

3.3.2 PROCESSO DE IMPLEMENTAÇÃO

Para a utilização da API *aglets* ou somente AAPI foi necessário fazer o importação das classes previamente definidas na AAPI, conforme ilustrado no quadro 2. A AAPI é definida através de suas classes e interfaces fundamentais e dos modelos de mensagens e eventos que implementa. Através do comando *import com.ibm.aglet.**, foram carregadas para o sistema as funcionalidades referentes à classe aglet utilizadas nos códigos-fontes que serão analisados. Na linha 5, tem-se a linha de comando *public class PeerAglet extends Aglet{*, onde se define que está criando a classe *PeerAglet* que estende as funcionalidade do objeto *Aglet* da AAPI. Para utilização do protocolo de comunicação e transferência *Agent Tranfer Protocol* (ATP), protocolo este utilizado pelos objetos do tipo *Aglet*, é necessário também importar a biblioteca *java.net.**, pois o protocolo ATP usa as mesmas funções do protocolo *TCP/IP*.

```
import com.ibm.aglet.*;
import java.io.File;
import java.net.*;
import java.util.Hashtable;
public class PeerAglet extends Aglet{
    public Peer Tela;
    public URL Servidor;
    public AgletProxy msgProxy,ServidorProxy,proxy;
    public URL destino,peer;
    public AgletID aid,serveraid,peeraid;
    public Object repli;
    public boolean servidor;
    private Hashtable tabela;
```

Quadro 2 –Importação da AAPI para utilização da classe *Aglet*.

Ainda sobre o quadro 2, foram declarados:

a) variáveis do tipo *Tela* que são responsáveis pela interface do aplicativo *Peer;*

b) variável do tipo URL, para o endereço ATP;

c) variáveis do tipo *AgletProxy* objeto que armazena todas as informações sobre os *Aglets*;

d) variáveis do tipo *AgletID* que recebe os *IDs*, que são identificadores únicos para

cada *Aglet* instanciado;

e) variável do tipo *Object* que irá receber o retorno de uma mensagem;

f) variável do tipo *boolean* que é o indicador do *status* do servidor;

g) por último uma variável do tipo *HashTable*, que guarda uma coleção de objetos do tipo *AgletProxy* e tem como chave, objetos do tipo *AgletID*.

No quadro 3 pode-se notar a instanciação de um objeto do tipo *URL*, através do comando *destino = new URL("atp://alexandre:12000/")*, onde destino é o objeto instanciado e *new URL* é o tipo do objeto a ser instanciado, *"atp://alexandre:12000/"*, é o objeto que está sendo instanciado. O endereço do servidor vai estar em um endereço fixo.

O comando *peer = this.getCodeBase()* da linha 5, inicializa a variável *peer* com o endereço do objeto que esta executando este comando. O comando *AgletProxy proxy = getAgletContext().createAglet(getCodeBase(),"AgletMsg",null)* da linha 12 fica responsável pela instanciação de um objeto do tipo *AgletProxy* onde passou-se como parâmetros o endereço a ser instanciado, o *Aglet* a ser instanciado é *null*, passado por parâmetro por significar ser um novo objeto.

```
try{
    //destino recebe um undereço ATP
    destino = new URL("atp://alexandre:12000/");
    // objeto peer do tipo URL recebe Endereço ATP onde esta instanciado este objeto PeerAglet
    peer = this.getCodeBase();
}catch (MalformedURLException e){
    //Erro de formação do endereço ATP
    envia("Erro de conexãoPeerAglet/run() "+e.getMessage());
}
try{
    //Instanciação do objeto AgletMsg
    AgletProxy proxy = getAgletContext().createAglet(getCodeBase(),"AgletMsg",null);
    //Variavel do tipo AgletID recebe o identificador do objeto criado
    aid = proxy.getAgletID();
    //objeto msgProxy recebe a referencia ao objeto dispachado para o destino,
    //onde destino é um endereço ATP
    msgProxy = proxy.dispatch(destino);
```

Quadro 3 – Instanciação dos objetos.

Tem-se ainda a instanciação dos objetos do tipo *AgletID* que recebe o identificador do objeto *AgletProxy* que fora criado, e por último cita-se a inicialização de um objeto do tipo

AgletProxy pois o objeto referenciado no comando já foi instanciado e através do comando *msgProxy = proxy.dispatch(destino)*, o objeto *msgProxy* recebe o novo *proxy* do objeto *proxy*, pois através do comando *dispatch(Destino)*, foi enviado o objeto para o endereço *destino* e ao mudar de ambiente o *aglet* altera as informações do seu *proxy*, como o endereço por exemplo. Foi então enviado um objeto do tipo *Agletmsg* até o endereço do servidor, que é conhecido e fixo, para pegar o numero de identificação do servidor.

Pode-se no quadro 4, verificar através do comando *repli = msgProxy.sendMessage (new Message("Hello"))*, a utilização de envio de uma mensagem simples, pois o conteúdo da mensagem é simplesmente *"Hello"*, é uma mensagem de comunicação do *peer* com o objeto *AgletMsg*, após ter sido enviado ao contexto do servidor, executa o comando que envia uma mensagem para todos os *Peers* daquele contexto. Pode-se ainda perceber a utilização da replicação da mensagem no comando *serveraid = (AgletID) repli* da linha 4, onde o *Peer* que enviou a mensagem espera por uma replicação à mensagem.

```
//Envio de mensagem ao objeto AgletMsg comando para o Aglet pegar o ID do servidor
repli = msgProxy.sendMessage(new Message("Hello"));
//variavel recebe a replicação a mensagem enviada, sendo ela o ID do servidor
serveraid = (AgletID)repli;
////com posse do endereço ATP e do ID do servidor pode-se agora pegar o proxy dele
ServidorProxy = getAgletContext().getAgletProxy(destino,serveraid);
//em caso de chegar até aqui é sinal que ocorreu tudo bem para pegar o proxy do servidor
servidor=true;
```

Quadro 4 –Comunicação simples do *Peer* com o *AgletMsg*.

Após o *peer* ter recebido do *AgletMsg* o identificador do servidor, tem-se possibilidade de pegar o *proxy* do servidor de nomes, através do comando *ServidorProxy = getAgletContext().getAgletProxy(destino,serveraid)* da linha 6, obtém-se então o *proxy*, os parâmetros do comando são o endereço onde o *aglet* que se quer obter o *proxy* está, que é um endereço fixo para este protótipo, e o outro parâmetro é o *id* do *aglet* a se obter o *proxy*.

No quadro 5, tem-se o uso do comando *subscribeMessage("Pega ID")*, que é responsável por registrar uma mensagem *multicast*, por que a mensagem esta sendo enviada

não através do *proxy* do *AgletServidor*, pois ainda não se sabe o *proxy*, mas sim para todos os aglets do ambiente, mas para isto o *aglet* deve registrar a mensagem para poder receber, pois se enviado a mensagem para todos aglets do ambiente e nenhum *aglet* do ambiente tenha esta mensagem registrada, nenhum *aglet* do ambiente vai receber a mensagem.

```
public void onCreation(Object init){
    //autoriza recebimento da mensagem do tipo Pega ID
    subscribeMessage("Pega ID");
    //Instancia Objeto do tipo servidor
    Servidor Tela = new Servidor();
```

Quadro 5 – Utilização do *Subscribemessage*.

Novamente citando o quadro 5, tem-se na linha 3, o exemplo da utilização do comando no servidor, onde é feito o registro de uma mensagem *multicast*, mensagem esta enviada pelo objeto *AgletMsg* ao chegar ao contexto do servidor de nomes, para que o servidor envie seu *id*.

No quadro 6, mostra-se a utilização do envio de mensagem *multicast*, situação esta implementada no objeto *AgletMsg*, pois ao ser enviado ao contexto do servidor de nomes, o *AgletMsg* não tem como enviar uma mensagem diretamente ao servidor de nomes, por não ter ainda o *proxy* do servidor de nomes. Através do comando *ReplySet replies = getAgletContext().multicastMessage (message)*, visualizado na linha 2, então tem-se o envio da mensagem *multicast*.

```
//por não saber o proxy do servidor envia para todos os aglets executando neste contexto
ReplySet replies = getAgletContext().multicastMessage(message);
//espera pela resposta
while (replies.hasMoreFutureReplies()){
    FutureReply future = replies.getNextFutureReply();
    try{
        //variavel servidor recebe ID do servidor
        servidor = (AgletID) future.getReply();
        //é respondido a mensagem Hello contendo o ID do Servidor
        msg.sendReply(servidor);
    }catch(Exception e){
        System.out.println("AgletMsg não enviou "+e.getMessage());
    }

}
//termina a função deste aglet e é exterminado
dispose();
```

Quadro 6 – Envio da mensagem *multicast.*

Pode-se ainda citar o uso do comando *replies.hasMoreFutureReplies()*, mostrado na

linha 4, como condição do bloco *while* que o fará ficar em *loop* até receber resposta de todos os *aglets* que receberem a mensagem *multicast*, para este caso, somente o *aglet* servidor de nomes é que teve o registro de uma mensagem *multicast*. Tem-se também na linha 5, a utilização do comando *FutureReply future = replies .getNextFutureReply()*, que de forma assíncrona recebe a replicação à mensagem *multicast* enviada.

No comando *msg.sendReply(servidor)*, nota-se uma replicação a uma determinada mensagem *msg* enviando então desta forma o *id* do servidor ao *aglet* que enviou o objeto *AgletMsg*. Por fim usa-se do comando *dispose()*, para terminar com um *aglet*, nesta utilização do comando, após ter enviado o *id* do servidor de nomes ao *peer* que enviou o *AgletMsg*, termina a função do *AgletMsg*.

No quadro 7, tem-se um exemplo de utilização de mensagem que tem como parâmetro duas variáveis do tipo *String*, formando assim uma mensagem composta por duas variáveis do tipo *String*, que pode ser visualizada através do comando da linha 3, *new Message("Conectar",peer.toString()+""+peeraid.toString())*, onde o *peer* de posse do *proxy* do servidor de nomes envia a ele um pedido de conexão, que é na verdade um pedido para que este *peer* seja incluído na lista de *peers* do servidor, ficando desta forma disponível para os outros *peers*.

```
//Envia uma mensagem agora ao servidor de nomes enviando a msg Conectar com o endereco
//e o ID
Object Reply = ServidorProxy.sendMessage(new Message("Conectar",peer.toString()+" - "+peeraid.toString()));
//imprime resposta do servidor
System.out.println((String) Reply);
```

Quadro 7 – Pedido de conexão do *peer* ao servidor de nomes.

Após ter sido incluído na lista de *peers* do servidor de nomes, o *peer* envia uma mensagem ao servidor de nomes para que seja atualizada a sua lista de *peers*, conforme quadro 8. A atualização da lista de *peers* conectados contido no *peer*, se da à cada inserção ou exclusão de um *peer* na lista de *peer* do servidor de nomes, desta forma as listas ficam

sincronizadas.

```
//envia msg para o servidor para enviar a lista de peers conectados a ele
Reply = ServidorProxy.sendMessage(new Message("Atualizar",peer.toString()+" - "+peeraid.toString()));
//imprime resposta do servidor
System.out.println((String) Reply);
```

Quadro 8 – Pedido de atualização da lista de *peers* do *peer*.

O servidor de nomes após ter recebido a mensagem de conexão do *peer* deve incluí-lo na sua lista de *peers*. Isto ocorre nos comandos do quadro 9, onde tem-se os seguintes processos:

a) trata-se a *string* recebida na mensagem de conexão do *peer*, pegando-se o *id* e o endereço do *peer*, pode-se visualizar este processo através dos comandos da linhas 6 e 9;

b) instancia-se o *proxy* do *peer* que esta se conectando ao servidor de nomes, visualizado através do comando da linha 12;

c) insere-se o *proxy* na tabela de *peers* do servidor de nomes, colocando em uma tabela *Hash*, que tem como chave o *id* do *peer*.

```
//função que insere peer na tabela de peers, recebe como parametro os dados numa string só
public void insere(String msg){
    //declaração de variaveis
    String peerid;
    //tratamento da string
    peerid = msg.substring(msg.indexOf(" - ")+3);
    try{
        //Instanciação da variaveis
        URL adress= new URL(msg.substring(0, msg.indexOf(" - ")));
        AgletID Ident = new AgletID(peerid);
        //pega o proxy do peer que esta se conectando
        proxy = getAgletContext().getAgletProxy(adress,Ident);
    }catch (Exception e){
        envia("Erro na instanciação do objeto proxy AgletServidor/insere");
    }
    //insere dados do peer na tabela
    Object val = tabela.put(peerid,proxy);
    //chama a função refresh que dá um refresh no JTextArea da tela do servidor
    refresh();
    //manda para o peer a tabela de proxy dos peers conectados ao servidor
}
```

Quadro 9 – Inserção do *peer* na tabela de *peers* do servidor de nomes.

Após ter-se inserido o *peer* manda-se a lista de *peers* à todos os *peers* da lista, através dos comandos do quadro 10, onde pode-se perceber novamente o tratamento da *string* que veio do *peer*. Após isso verifica-se se tem algum *peer* inserido na tabela de *peers*, se tiver

então percorre-se a tabela de *peers*, enviando a todos os *peers* contidos na tabela a própria tabela de *peers*, uma mensagem que é composta pela *string "InserePeer"*, que identifica a mensagem, e ainda a tabela contendo todos os *peers* conectados ao servidor de nomes. Neste caso foi utilizado o envio de mensagem assíncrona, através do comando *proxyII.sendFutureMessage(new Message ("InserePeer",tabela))*, não foi possível enviar a mensagem de forma síncrona, que só é recebida e respondida, quando há um sincronismo entre os *aglets*, se o *aglet* que recebe a mensagem, estiver executando outro processo e não estiver sincronizado para receber a mensagem, a mensagem não é recebida e o *aglet* que enviou a mensagem espera por um retorno, travando assim a aplicação.

```
public void mandaInserir(String msg){
    //declaração de variaveis
    String peerid;
    //tratamento da string
    peerid = msg.substring(msg.indexOf(" - ")+3);
    AgletID Ident = new AgletID(peerid);
    try{
        URL adress= new URL(msg.substring(0, msg.indexOf(" - ")));
        //Verifica se tem algum peer conectado
        if (tabela.size()>0){
            //pega os elementos da tabela
            for (Enumeration enum = tabela.elements();enum.hasMoreElements();){
                try{
                    envia("Enviando msg para inserir peer aos peers");
                    //Pega o proxy na tabela
                    proxyII = (AgletProxy)enum.nextElement();
                    //enviar msg para inserir peer na lista de peer´s
                    proxyII.sendFutureMessage(new Message("InserePeer",tabela));
                }catch(Exception e){
                    envia("Erro no AgletServidor/mandaInserir -->"+e.getMessage());
                }
            }
        }
    }catch(Exception e){
        envia("erro ao enviar a lista em AgletServidor/mandaInserir -->"+e.getMessage());
    }
```

Quadro 10 – Atualização da lista de *peers* utilizada nos *peers*.

No quadro 11, vê-se a utilização do evento *onDisposing*, implementado para percorrer a lista de *peer* e enviar uma mensagem avisando aos *peers* que o servidor de nomes foi desativado. A partir deste momento, os *peers* continuam com a lista de *peers*, pode-se ainda continuar trocando mensagens , porém já não é garantida a consistência das informações da lista.

```
//quando o servidor for desconectado

public void onDisposing(){
    //Verifica se tem algum peer conectado
    if (tabela.size()>0){
        //pega os elementos da tabela
        for (Enumeration enum = tabela.elements();enum.hasMoreElements();){
            try{
                envia("Enviando mensagem de desligamento para os peers");
                //Pega o proxy na tabela
                proxyII = (AgletProxy)enum.nextElement();
                //Envia mensagem avisando que o servidor esta sendo desconectado
                proxyII.sendMessage(new Message("ServidorTchau"));
            }catch(Exception e){
                envia("Erro no AgletServidor/onDisposing -->"+e.getMessage());
            }
        }
```

Quadro 11 – Uso do evento *onDisposing* no servidor.

No quadro 12, tem-se a utilização do evento *handleMessage*, responsável por receber toda comunicação entre os objetos *aglets*, visualiza-se também, através do comando da linha 5, a utilização dos argumentos enviados em uma mensagem composta .

```
    public boolean handleMessage(Message msg){
    //variavel aid pega o ID do aglet peer
    aid = getAgletID();
    if (msg.sameKind("Hello")){//implementar caso de o servidor ser terminado
        Tela.display((String)msg.getArg());
    }else if (msg.sameKind("Pega ID")){
        //se solicitaram o ID dar uma resposta a mensagem com o ID do Servidor
        msg.sendReply(aid);
    }else if (msg.sameKind("Conectar")){
        //se houve um pedido de conexão inserir na lista de peers
        insere((String)msg.getArg());
        msg.sendReply("\n\nok inserido na tabela do servidor\n\n");
    }else if (msg.sameKind("Desconectar")){
        //Peer foi desconectado, excluir da lista de peers
        Exclui((String)msg.getArg());
    }else if (msg.sameKind("Atualizar")){
        //Peer pedepara atualizar listade peers
        mandaInserir((String)msg.getArg());
        msg.sendReply("\n\nok Atualizado\n\n");
```

Quadro 12 – Utilização do evento *handleMessage* no servidor.

Como no Exemplo do comando *insere((String)msg.getArg())*, onde é chamado o método *insere()* passando como parâmetro os argumentos da mensagem que é o endereço do *aglet peer* que está se conectando e seu *id*, porém passados como uma única *String*.

No quadro 13, tem-se o método responsável pela verificação dos arquivos XML do *aglet peer*, para este trabalho definiu-se um diretório padrão onde serão armazenados os arquivos XML, que serão à base de pesquisa dos aglets. No método *VerificaArquivos()*, tem-se a verificação de todos arquivos contidos no diretório, sendo então armazenados os nomes

somentes dos arquivos XML através do comando visualizado na linha 6, *lista[i].substring(lista[i].indexOf(".")+1).compareToIgnore Case("XML")== 0*, verificando assim se a extensão do arquivo é “.XML”, para então incluído no *array* Bases.

```
public void VerificaArquivos(){
    File nome = new File("C:\\Java\\Arquivos");
    if (nome.isDirectory()){
        String lista[] = nome.list();
        for (int i = 0;i < lista.length; i++){
            if (lista[i].substring(lista[i].indexOf(".")+1).compareToIgnoreCase("xml")== 0){
                //inclusao do nome dos arquivos xml no Array Bases
                Bases[i] = lista[i].substring(0, lista[i].indexOf("."));
            }
        }
    }else{
        System.out.println("Não é um diretorio\n");
    }
}
```

Quadro 13 – método *VerificaArquivos*.

Após ter verificado a existência do arquivo XML no diretório padrão, notifica-se todos os *peers* de sua lista de *peers* sobre a existência arquivo XML, para que nos outros *peers* tenham essa informação e mostre através do objeto *TelaPeer*, quantas bases estão disponíveis para pesquisa. A partir deste ponto qualquer *peer* pode consultar a base disponível em qualquer *peer*. Para consultar deve-se pressionar o botão pesquisar do objeto *TelaPeer* , onde apresenta um objeto do tipo *telaPesquisa*, com campos de acordo com os atributos dos elementos dos arquivos XML. Após preencher os campos que são relevantes para pesquisa, deve-se pressionar o botão *Pesquisar*, com os dados digitados pelo usuário, instancia-se um objeto do tipo *Carro*.

Pode-se visualizar a implementação da classe *Carro* no quadro 14, composta dos atributos *modelo*, *marca*, *motor*, *cor*, *combustível*, *preço* e *ano*. Visualiza-se ainda os métodos atribuídos ao objeto do tipo *Carro*, métodos responsáveis pela disponibilização dos atributos. Tendo-se um objeto *Carro* proveniente da tela pesquisa, é instanciado um objeto do tipo *AgletConsulta*, enviado para o ele o objeto do tipo *Carro* da pesquisa, finalizada esta etapa, é enviado para o ambiente de cada *peer* da lista um clone do *AgletConsulta* e a partir deste

momento dâ-se o processamento distribuído e a comunicação *peer-to-peer*.

```
private String modelo, marca, motor, cor, combustivel, preco, ano;
public Carro(String mod, String mar, String mot, String c, String combus, String pre, String a){
    this.modelo = mod;
    this.marca = mar;
    this.motor = mot;
    this.cor = c;
    this.combustivel = combus;
    this.preco = pre;
    this.ano = a;
}
public String getModelo(){
    return this.modelo;
}
public String getMarca(){
    return this.marca;
}
public String getMotor(){
    return this.motor;
}
public String getCor(){
    return this.cor;
}
public String getComb(){
    return this.combustivel;
}
public String getPreco(){
    return this.preco;
}
public String getAno(){
    return this.ano;
}
```

Quadro 14 – Implementação da classe *Carro*.

No objeto *AgletConsulta*, tem-se implementado todo o processo de pesquisa nos arquivos XML, para isto deve-se ter um *Parser*, objeto este responsável pela leitura e interpretação dos arquivos XML.

Para implementação do *parser* mostrado neste trabalho, foi utilizado a biblioteca *SAX* importando as seguintes classes contidas na própria biblioteca java, *javax.XML.parsers.**, *org.XML.sax.** e *org.XML.sax.helpers.DefaultHandler*. Pode-se ver através do quadro 15, onde pelo comando *SAXParser parser = SAXParserFactory.New Instance().newSAXParser()*, dá-se a instanciação do *parser* dos arquivos XML. Após a instanciação do *parser*, ocorre uma conexão com o arquivo XML, através do comando da linha 14, pode-se visualizar esta conexão através do comando *InputSource input = new InputSource(arquivo)*, onde *arquivo* é o caminho completo para o arquivo XML.

```
//Diretório padrão dos arquivos
String arquivo = "C:\\Java\\Arquivos\\carros.xml";
try {

    ////////////////// inicia o parser ////////////////////////

    // cria a factory e o parser
    SAXParser parser = SAXParserFactory.newInstance().newSAXParser();

    // abre a conexao com o arquivo, buffer de 1 mega
    InputSource input = new InputSource(arquivo);

    // inicia o parsing
    parser.parse(input, new XMLHandler());
    System.out.println("Terminou!!");

} catch (ParserConfigurationException ex) {
    // se ocorrer um erro de parsing
    ex.printStackTrace();
} catch (SAXException ex) {
    // se ocorrer um erro de parsing
    ex.printStackTrace();
} catch (IOException ex) {
    // se ocorrer um erro de io
    ex.printStackTrace();
}
```

Quadro 15 – Implementação do *parser* dos arquivos XML.

Através do comando da linha 14 *parser.parse(input, new XMLHandler())*, é processado o arquivo XML, passando como parâmetro a conexão com o arquivo XML e um processador de arquivos XML. Pode-se visualizar ainda no quadro 15, tratamentos de exceções nas linhas 16, para erro de configuração do *parser*, na linha 19, para erro ocorridos na execução dos comandos da *api sax* e 21, onde trata erros de entrada ou saída. Se tudo ocorrer de forma correta na leitura do arquivo XML, no final da leitura é enviado uma mensagem de término ao usuário, através do comando da linha 15.

Como processador do arquivo XML criou-se uma classe que extende métodos e atributos da classe *XMLHandler*, porém com modificações para adequar-se ao contexto do trabalho, sendo possível visualizar a implementação no quadro 16, onde percorre-se o arquivo XML, verificando-se qual atributo do elemento *carro* do arquivo XML esta sendo lido, após verificado de que atributo se trata coloca-se o valor do atributo pesquisado no atributo de mesma nomenclatura do objetos do tipo *Carro*.

```
if (tag.compareToIgnoreCase("Carro")==0){
    XMLHandler.this.novo();
}
if ((tag.compareToIgnoreCase("modelo")==0)){
    modelo=valorAtual.toString().trim();
}
if (tag.compareToIgnoreCase("marca")==0){
    marca=valorAtual.toString().trim();
}
if (tag.compareToIgnoreCase("motor")==0){
    motor=valorAtual.toString().trim();
}
if (tag.compareToIgnoreCase("cor")==0){
    cor=valorAtual.toString().trim();
}
if (tag.compareToIgnoreCase("combustivel")==0){
    combustivel=valorAtual.toString().trim();
}
if (tag.compareToIgnoreCase("preco")==0){
    preco=valorAtual.toString().trim();
}
if (tag.compareToIgnoreCase("ano")==0){
    ano=valorAtual.toString().trim();
    obj = new Carro(modelo, marca, motor, cor, combustivel, preco, ano);
}
```

Quadro 16 – Implementação da instanciação dos elementos do arquivo XML .

Pelo quadro 17, pode-se verificar a estrutura do arquivo XML, onde cada elemento é um carro da base de dados e estes elementos contêm atributos que o qualificam. Para este trabalho, como exemplo, temos o arquivo XML *carros.XML*, onde o primeiro atributo tem o nome de *atributo* e seu valor é uma lista dos atributos de cada elemento. Nas linhas 2 à 10, tem-se um elemento *carro*, composto pelos atributos *modelo*, *marca*, *motor*, *cor*, *combustível*, *preço e ano*, ou seja, mesmos atributos da classe *Carro*, usada neste trabalho.

```
<Atributos>marca, motor, cor, combustivel, preço, ano</Atributos>
<carro>
   <Modelo>Gol</Modelo>
   <Marca>Wolkswagen</Marca>
   <Motor>1.6</Motor>
   <Cor>Branco</Cor>
   <Combustivel>Gasolina</Combustivel>
   <preco>15.000</preco>
   <Ano>2004</Ano>
</carro>
<carro>
   <Modelo>Gol</Modelo>
   <Marca>Wolkswagen</Marca>
   <Motor>1.6</Motor>
   <Cor>Branco</Cor>
   <Combustivel>Gasolina</Combustivel>
   <preco>15.000</preco>
   <Ano>2005</Ano>
</carro>
<carro>
   <Modelo>Gol</Modelo>
   <Marca>Wolkswagen</Marca>
   <Motor>1.6</Motor>
```

Quadro 17 – Estrutura do arquivo XML.

Atentando que a ordem dos atributos devem ser sempre a mesma para não haver inconsistência nos resultados da pesquisa.

No quadro 18, implementa-se o processo de pesquisa no arquivo XML, onde após a instanciação do objeto *Carro*, composto por um elemento do arquivo XML, que verifica se este objeto atende aos itens da pesquisa efetuada anteriormente pelo usuário. Para isto percorre-se os atributos do objeto do tipo *Carro* instanciado com os atributos do elemento do arquivo XML comparando com os atributos do objeto do tipo *Carro* instanciado com os dados da pesquisa, para comparar os atributos dos dois objetos.

```
boolean igual=true;

if (Pesquisa.getModelo().compareToIgnoreCase("")!=0){
    if (obj.getModelo().compareToIgnoreCase(Pesquisa.getModelo())!= 0){
        igual=false;
    }
}
if (Pesquisa.getMarca().compareToIgnoreCase("")!=0){
    if (obj.getMarca().compareToIgnoreCase(Pesquisa.getMarca())!=0){
        igual=false;
    }
}
if (Pesquisa.getMotor().compareToIgnoreCase("")!=0){
    if (obj.getMotor().compareToIgnoreCase(Pesquisa.getMotor())!=0){
        igual=false;
    }
}
if (Pesquisa.getCor().compareToIgnoreCase("")!=0){
    if (obj.getCor().compareToIgnoreCase(Pesquisa.getCor())!=0){
        igual=false;
    }
}
if (Pesquisa.getComb().compareToIgnoreCase("")!=0){
    if (obj.getComb().compareToIgnoreCase(Pesquisa.getComb())!=0){
        igual=false;
    }
}
if (Pesquisa.getPreco().compareToIgnoreCase("")!=0){
    if (obj.getPreco().compareToIgnoreCase(Pesquisa.getPreco())!=0){
        igual=false;
    }
}
if (Pesquisa.getAno().compareToIgnoreCase("")!=0){
    if (obj.getAno().compareToIgnoreCase(Pesquisa.getAno())!=0){
        igual=false;
    }
}
if(igual){
    listacarros.Adiciona(new Carro(modelo, marca, motor, cor, combustivel, preco, ano));
    System.out.println("\n"+obj.getCarro());
}
```

Quadro 18 – Processamento do elemento do arquivo XML.

A pesquisa pode ser construída a partir da seleção de um ou mais atributos, a idéia geral do processamento é criar uma variável do tipo *boolean* que inicialmente recebe valor

verdadeiro e testa-se para cada atributo do objeto *Pesquisa* diferente de vazio, a desigualdade do valor do atributo entre os dois objetos, desprezando se o valor do atributo esta em caixa alta ou não. Se os atributos forem diferentes então a variável *boolean* recebe valor falso, identificando que algum dos atributos do elemento *carro* do arquivo XML é diferente do atributo do objeto *Pesquisa*.

Em caso de ser lido o último atributo e a variável *boolean* ainda possuir valor verdadeiro é sinal que este elemento do arquivo XML, atende aos atributos da pesquisa. Sendo incluído em uma lista de objetos do tipo *Carro*.

Lista esta definida pelo código apresentado no quadro 19, onde é instanciado um objeto do tipo *Hashtable*, e armazena-se os elementos que atenderem aos atributos da pesquisa, para manusear esta lista implementou-se os métodos :

a) *Adiciona()*, responsável por incluir um novo objeto *Carro* na lista;

b) o método *getLista()*, que é encarregado por enviar a lista de objetos;

c) método *regs()*, que apresenta o numero de registros que a lista contem ;

d) e por último o método *vazio()* que retorna um *boolean*, que identifica se a lista de *Carros* contem algum registro.

```
private Hashtable listacarros;
public listaCarros(){
    listacarros = new Hashtable();
}
public void Adiciona(Carro car){
    listacarros.put(car.getCarro(),car);
}
public Hashtable getLista(){
    return listacarros;
}
public int regs(){
    return listacarros.size();
}
public boolean vazio(){
    return listacarros.isEmpty();
}
```

Quadro 19 – Implementação da classe *listacarros.*

Após ter percorrido e comparado todos elementos do arquivo XML, tem-se a lista com

o resultado da busca, então cada *peer* envia ao *peer* que solicitou a pesquisa, a lista com os resultados da pesquisa no seu arquivo XML. O *peer* que está recebendo a resposta à pesquisa que solicitou, atualiza a tela de resultados a cada recebimento de lista de algum *peer*, até o recebimento da lista de resultados da busca de todos os peers, que constam na lista de *peer*.

3.3.3 OPERACIONALIDADE DA IMPLEMENTAÇÃO

Este tópico aborda o teste de operacionalidade da implementação, apresentando através de um estudo de caso que compreende a pesquisa em uma base de dados XML. Para tanto montou-se uma rede de dois computadores, um que foi referenciado com o endereço atp *atp://Alexandre:12000/,* e outro com o endereço *atp://Matheus:13000/.*

Para as bases de dados XML definiu-se o seguinte, em *atp://Matheus:13000/,* colocou-se uma base contendo os elementos do quadro 20, onde os elementos mostram-se na forma de texto, e cada linha do texto representa um elemento carro com seus atributos na base de dados XML. Base esta nomeada de *Carros.XML* e gravado no diretório *C:\Java\Arquivos,* este diretório é padrão para armazenar os arquivos XML em todo computador que utiliza o aplicativo *peer* desenvolvido neste trabalho.

```
Carros
        Golf, Wolkwagen,2.0, Branca, Gasolina, 29000, 1999
        Gol, Wolkswagen, 1.8, Preta, Gasolina, 16000, 1997
        Gol, Wolkswagen, 1.6, Vermelha, Gasolina, 11000, 1995
        Gol, Wolkswagen, 1.8, Branca, Alcool, 23000, 2001
        Golf, Wolkwagen,1.8, Branca, Gasolina, 25000, 1999
        Fusca, Wolkswagen, 1.3, Azul, Gasolina,  3000, 1987
        Fusca, Wolkswagen, 1.4, Verde, Alcool,  2500, 1975
        Fusca, Wolkswagen, 1.3, Branca, Gas,  4000, 1990
        Saveiro, Wolkwagen, 1.6, Branca, Alcool, 10000, 1995
        Golf, Wolkwagen,2.0, Azul, Alcool, 35000, 2002
        Pampa, Ford, 1.8, Verde, Gasolina,23000, 1991
        Pampa, Ford, 1.6, preta, Gasolina,35000, 2002
        Pampa, Ford, 1.4, Branca, Alcool,20000, 1990
        Santana, Wolkwagen, 1.8, Azul, Gasolina, 20000, 1998
        Santana, Wolkwagen, 2.0, Branca, Gasolina, 15000, 1996
        S10, Wolkswagen, 4.1, Branca, Gasolina, 45000, 1999
/carros
```

Quadro 20 – Base de dados de *atp:Matheus:13000/,* em forma de texto.

Já no computador *atp://alexandre:12000/,* tem-se no diretório padrão um arquivo XML com os elementos do quadro 21, com a mesma representação descrita para os elementos

do quadro 20.

```
Carros
        Gol, Wolkswagen, 1.8, Branca, Gasolina, 17000, 1999
        Gol, Wolkswagen, 1.6, Vermelha, Gasolina, 11000, 1995
        Gol, Wolkswagen, 1.8, Branca, Alcool, 23000, 2001
        Fusca, Wolkswagen, 1.3, Branca, Gasolina,  3000, 1987
        Fusca, Wolkswagen, 1.4, Verde, Gasolina,  2700, 1975
        Fusca, Wolkswagen, 1.3, Branca, Gas,  4000, 1990
        Saveiro, Wolkwagen, 1.6, Branca, Gasolina, 10000, 1995
        Golf, Wolkwagen,2.0, Azul, Alcool, 35000, 2002
        Golf, Wolkwagen,2.0, Prata, Gasolina, 45000, 2005
        Pampa, Ford, 1.8, Branca, Gasolina,23000, 1991
        Pampa, Ford, 1.6, preta, Gasolina,35000, 2002
        Pampa, Ford, 1.4, Branca, Alcool,20000, 1990
        Santana, Wolkwagen, 1.8, Azul, Gasolina, 20000, 1998
        Santana, Wolkwagen, 2.0, Vermelha, Gasolina, 18000, 1998
        S10, Wolkswagen, 4.1, Branca, Gás, 35000, 2000
/Carros
```

Quadro 21 – Base de dados de *atp://Alexandre:12000/*, em forma de texto.

Para utilização dos aplicativos descritos neste trabalho será necessário a instalação do aplicativo *jdk-1_5_0_02-windows-i586-p.exe*, que é um conjunto de bibliotecas necessárias para execução de aplicativos escritos em Java, e após instalado o *jdk*, deve-se instalar o pacote de biblioteca responsável pela implementação dos *aglets*, bem como o aplicativo *Tahiti*, que é responsável pela configuração do ambiente em que executa-se os *aglets*. Tendo instalado e configurado todo o ambiente de forma correta pode-se então utilizar os aplicativos desenvolvidos neste trabalho.

Após ter incluído uma base de dados XML no diretório padrão de cada computador, deve-se executar o aplicativo servidor de nomes em um dos computadores, para este trabalho é no computador *atp://Alexandre:12000/*, pois todo aplicativo *peer* que é executado, envia um objeto do tipo *agletmsg*, para este endereço *atp*, com a responsabilidade de pegar o número de identificação do servidor de nomes.

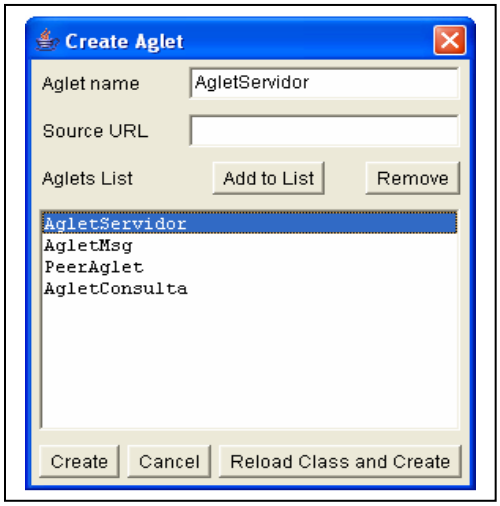
A execução dos dois aplicativos desenvolvidos neste trabalho dá-se através do *Tahiti*, que gerencia o ambiente e permite executar alguns comandos dos aglets, como: criar *aglets*, obter informações dos *aglets*, fechar o *aglet*, clonar, enviar e chamar de volta o *aglet*, porém neste trabalho utiliza-se somente o gerenciamento do ambiente e a função criar *aglet*, ficando o restante das funções a encargo do próprio aplicativo *peer*.

Na execução do aplicativo *Tahiti*, que deve ser via *prompt* através do comando "*agletsd –port*", onde *port* é o número da porta de execução do servidor de *aglets*, pede-se um *login*, onde como usuário usa-se *anonymous* e senha *aglets*. Após o usuário estar devidamente "logado", tem-se o aplicativo *Tahiti* e a função *Create* disponível para utilização, conforme pode-se visualizar no quadro 22, onde apresenta o aplicativo *Tahiti*, sem nenhum aglet em execução. O nome do host e o número da porta em que o *Tahiti* está executando, e o nome do usúario que efetuou o login no servidor, pode ser visto na barra de título da janela.

Quadro 22 – Apresentação do aplicativo *Tahiti*.

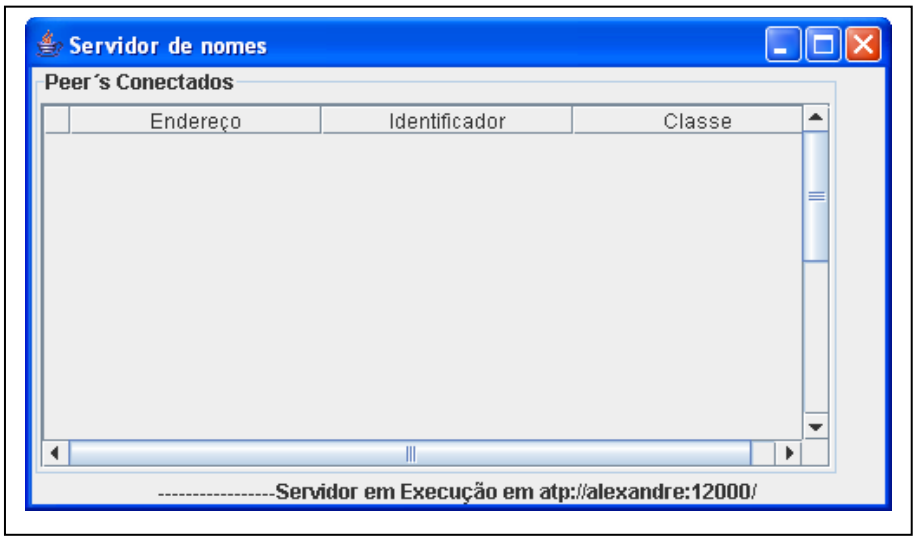
Pode-se a partir deste momento executar um *aglet*, pois para executar o aplicativo *Tahiti*, todo o ambiente deve estar configurado corretamente.

No computador *atp://Alexandre:12000/*, deve-se então executar o aplicativo servidor de nomes conforme descrito, por ser o endereço padrão para execução do servidor de nomes e todos *agletsPeer* enviam o objeto do tipo *Agletmsg* para este endereço para pegar o *id* do servidor de nomes, para isto pressiona-se o botão *Create* do aplicativo *Tahiti* e conforme o quadro 23, deve-se selecionar o *AgletServidor*, nome do protótipo do servidor de nome, da lista de *aglets* disponíveis desta tela de seleção.



Quadro 23 – Seleção do *aglet* a ser executado através do *Tahiti*.

Após isto tem-se então a execução do aplicativo servidor de nomes implementado neste trabalho, pode-se visualizar o aplicativo através do quadro 24, composto por uma tabela que apresenta os principais dados referentes aos *peer* conectados ao servidor de nomes, dados como endereço, número de identificação e classe a que cada *peer* conectado pertence. Tem-se ainda uma barra de status com mensagens do servidor ou sobre os *peers* conectados.

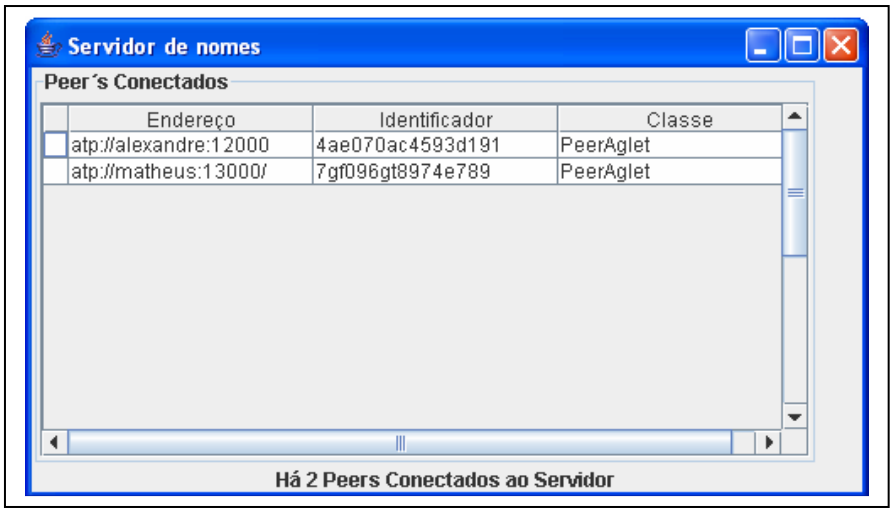


Quadro 24 – Execução do servidor de nomes.

Executou-se também no computador *atp://Alexandre:12000/*, da mesma forma pelo aplicativo *Tahiti*, o aplicativo *peer*.

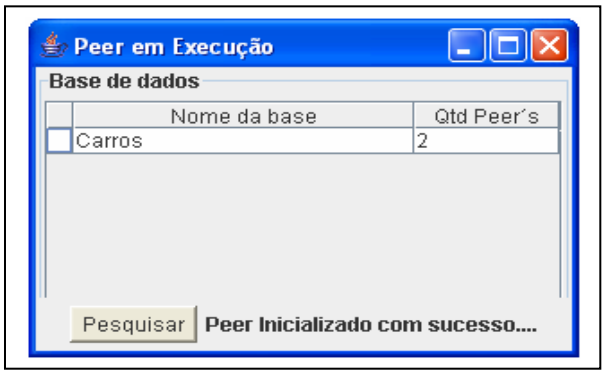
No computador *atp://Matheus:13000/*, executou-se através do *Tahiti* o aplicativo *peer*. Tendo então a apresentação do aplicativo servidor de nomes no computador

atp://Alexandre:12000/, com dois *peers* conectados, conforme apresenta o quadro 25, onde pode-se visualizar o endereço, identificador e classe dos aplicativos peers conectados ao servidor de nomes, e na barra de *status* a mensagem da quantidade de peers conectados ao servidor de nomes.



Quadro 25 – Apresentação do servidor de nomes com 2 *peers* conectados.

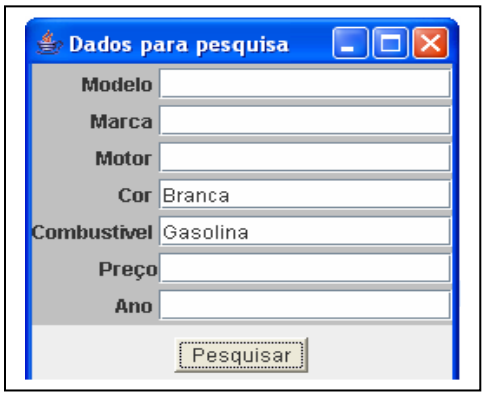
Os aplicativos *peers* apresentam-se conforme quadro 26, onde pode-se visualizar o nome da base XML e a quantidade de *peer* que tem a base disponível para pesquisa, neste caso dois *peers* com a base de dados *Carros.XML*.



Quadro 26 – Visualização dos aplicativos *peers*.

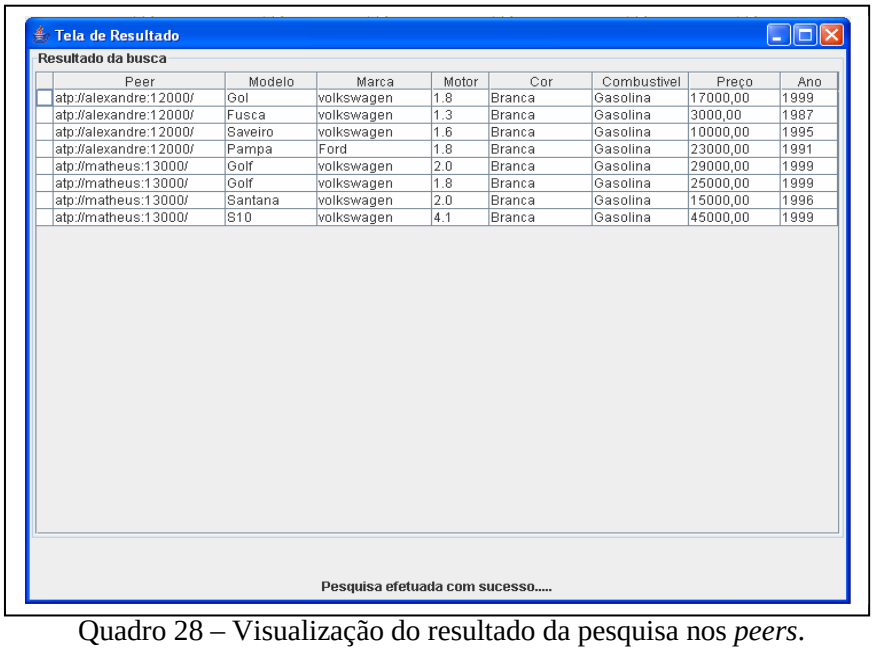
Nota-se ainda, no aplicativo *peers*, um botão para efetuar a pesquisa e uma barra de status com mensagens referente ao *peer*. Para efetuar uma pesquisa pressiona-se o botão Pesquisar, o qual apresenta uma tela conforme mostra o quadro 27, podendo então definir os atributos da pesquisa. Para este caso de uso foi definido que a pesquisa deve trazer os carro das bases de dados XML que estiverem cadastrados com a cor branca e o combustível sendo

gasolina.



Quadro 27 – Visualização da tela de pesquisa.

Definido os atributos da pesquisa, pressiona-se o botão Pesquisar, o qual dispara a pesquisa nos dois *peers* conectados e depois de terminado a pesquisa na base de dados dos *peers* tem-se no *peer* que solicitou a pesquisa o resultado da pesquisa conforme mostrada no quadro 28, contendo uma tabela com os seguintes atributos *peer*, um atributo na coluna nomeada *Peer*, que mostra o endereço da base XML que foi encontrado o carro, em seguida tem-se os atributos do carro, que são: modelo, marca, motor, cor, combustível, preço e ano. E visualiza-se no quadro 28, os carros pesquisados no computador *atp://Alexandre:12000/* e no computador *atp://Matheus:13000/*, de acordo com os atributos da pesquisa selecionadas anteriormente no quadro 27.



Quadro 28 – Visualização do resultado da pesquisa nos *peers*.

3.4 RESULTADOS E DISCUSSÃO

De acordo com os teste realizados para o estudo de caso apresentado, obteve-se bons resultados em todos os aspectos do protótipo, como no caso da utilização da API do java para leitura dos arquivos XML, pois verificou-se uma boa performance nos resultados dos testes.

Conforme tabela 1, onde constam os resultados obtidos na comparação na pesquisa realizada somente em um computador ou utilizando os protótipos desenvolvidos neste trabalho em um computador local e outro computador remoto.

Tabela 1 – Comparativo dos resultados em milisegundos.

registros	tempo	t/registro	2Peers	t/registro2peers	tempo ganho
500	156	0,312	101,4	0,2028	54,6
5000	281	0,0562	182,65	0,03653	98,35
25000	906	0,03624	588,9	0,023556	317,1
50000	1656	0,03312	1076,4	0,021528	579,6
100000	3390	0,0339	2203,5	0,022035	1186,5
500000	7797	0,015594	5068,05	0,0101361	2728,95

O valor da coluna registros refere-se a quantidade de registros dos arquivos XML, os valores das colunas *tempo* e *t/registro* são referentes a execução da pesquisa com um computador ao arquivo XML, os valores das colunas *2Peers* e *t/registro2Peers*, são resultados obtidos na leitura do arquivo XML com o uso dos protótipos desenvolvidos e a coluna *tempo ganho* representa a diferença entre o tempo gasto por um computador e o tempo gasto com os protótipos. Os valores das colunas *tempo*, *t/regis*, *2Peers*, *t/registro2Peers* e *tempo ganho*, são medidos em milisegundos. Para ficar mais fácil o entendimento dos dados visualizados na tabela 1, é feita a leitura da primeira linha da tabela.

Quando a leitura foi feita em um arquivo XML contendo 500 registros, tem-se os seguintes dados como resultado:

a) para um computador fazendo a leitura, o tempo total gasto para a leitura dos registros foi de 156 milisegundos, logo, a leitura por registro foi de 0,312 milisegundos;

b) para dois computadores utilizando os protótipos desenvolvidos neste trabalho, o tempo total gasto para leitura dos registros foi de 101,4 milisegundos e o tempo gasto por registro ficou em 0,2028 milisegundos.

Nos testes usando os protótipos o arquivo XML foi dividido pela quantidade de *peers*, envolvidos nos testes, que apresentam resultados melhores executando a mesma tarefa.

Podemos visualizar através da tabela 2, um gráfico comparativo entre o tempo gasto por registro em ambos os casos da pesquisa, onde a linha que contém os nodos quadrados referem-se ao tempo gasto por registros executados por um computador e a linha com os nodos triangulares representam o tempo gasto por registro na pesquisa efetuada utilizando os protótipos.

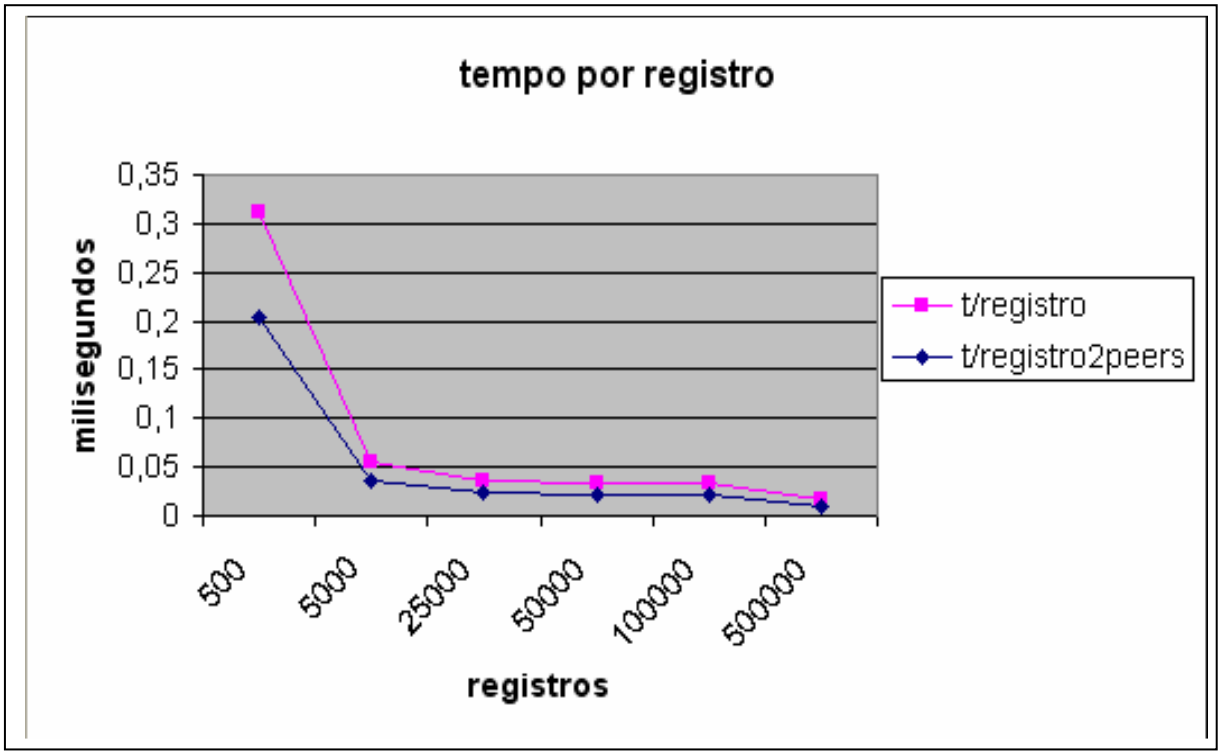


Gráfico 1 – Comparativo de tempos gastos por registros.

Na tabela 3, tem-se um gráfico que mostra a diferença entre o tempo total gasto na leitura de cada arquivo XML, para os testes realizados em um computador e nos testes

realizados utilizando um computador local e outro remoto, em uma rede utilizando os protótipos desenvolvidos neste trabalho. Chamou-se a linha resultado de *tempo ganho*, por que este é o tempo de processamento, que foi "economizado" no computador que fez a pesquisa na base de dados XML.

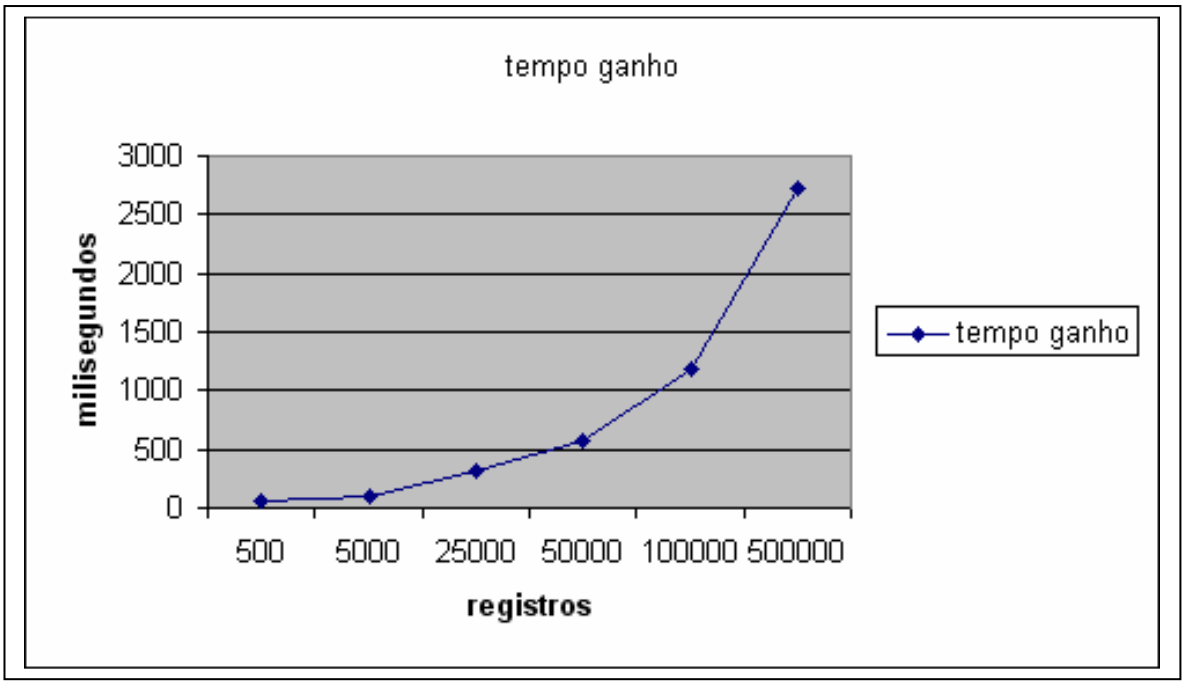


Gráfico 2 – Diferença do tempo total dos casos de teste.

4 CONCLUSÕES

Ao término do desenvolvimento do trabalho pode-se concluir que os resultados foram alcançados em relação aos objetivos previamente formulados. Foram implementados dois protótipos: um que realiza a tarefa de servidor de nomes gerenciando e distribuindo a lista de *peers* da rede para todo computador que, executando o protótipo do aplicativo *peer*, for adicionados à lista do servidor de nomes; outro que realiza a função de *peer*, conectando-se ao servidor de nomes, disponibilizando seu poder de processamento, trocando mensagens com outros *peers* e executando processamento em arquivos XML.

Também como especificado nos objetivos do trabalho, foram utilizados dois computadores de uma rede local, para formar o processamento distribuído, onde um computador executou o protótipo servidor de nomes e também de peer e outro computador executou o protótipo, efetuando somente o papel de *peer*.

Pode-se perceber durante o desenvolvimento do trabalho que o uso de agentes móveis está tornando-se, uma área bastante promissora de soluções e novos enfoques para várias aplicações de rede. Verificou-se ainda a importância do paradigma de agentes móveis dentro da área de sistemas distribuídos, bem como a utilização do conceito de processamento distribuído. Com a utilização destes paradigmas no desenvolvimento de aplicativos, pode-se ter a solução para o problema de falta de poder computacional, seja para aplicações que atualmente sejam tidas como inviáveis, ou para usuários comuns que necessitem de um poder computacional não disponível em seu computador, simplesmente.

Verificou-se através da análise dos tempos de processamento dos testes efetuados no protótipo do trabalho que, em uma rede de computadores onde há ociosidade nos computadores da rede, a utilização da ociosidade é uma forma de aumentar o poder computacional. Conforme nos testes efetuados, pode-se perceber que usando a ociosidade de

outra máquina e o uso do protótipo, obtém-se resultados melhores do que se o processamento ocorre-se em um único computador. Tendo desta forma um supercomputador virtual para utilização em aplicações que necessitem de grande poder de processamento.

Através dos resultados obtidos nos testes efetuados no protótipo, pode-se perceber que com o uso do protótipo em uma maquina local e outra remota houve uma diminuição do tempo gasto no processamento da pesquisa.

4.1 EXTENSÕES

De acordo com os conhecimentos adquiridos durante o desenvolvimento do trabalho tornou-se possível à análise de diversas possibilidades que podem ser sugeridas como trabalhos futuros, sendo elas:

d) adaptação do protótipo atual para a leitura de diversas bases de dados XML, de forma a moldar a consulta de acordo com a base escolhida para ser pesquisada;
e) desenvolvimento de um servidor de *aglets*, para não utilizar o aplicativo *Tahiti*, diminuindo assim a quantidade de Softwares, aplicativos, *APIs* e afins instalados para a utilização do protótipo;
f) desenvolver uma forma de balanceamento das cargas, levando em consideração o poder computacional de cada computador da rede.
g) testar o protótipo com um número de máquinas, que levem a alcançar os limites do protótipo.

REFERÊNCIAS BIBLIOGRÁFICAS

ARIDOR, Yariv; LANGE, Danny. **Agent Design Patterns: Elements of Agent Application Design**. In: Proceedings of the second international conference on autonomous agents (AGENTS ‘98). Springer Verlag, 1998. Disponível em: <http://www.moe-lange.com/danny/patterns.pdf>. Acesso em: 25 set. 2004.

BORCHARDT, Christiano M. **Desenvolvimento de um mecanismo gerenciador de transações para sistemas distribuídos.** 2002. 89 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

CERN. **The world's largest particle physics laboratory**. Genebra, [1993]. Disponível em: <http://public.web.cern.ch/Public/Welcome.html>. Acesso em: 25 set. 2004.

CLARKE, Ian. **The free network project.** [S.I.], 1999. Disponível em: < http://freenetproject.org/ >. Acesso em: 28 out. 2004.

COELHO, Alexandre Rodrigues. **Linguagens e protocolos de comunicação entre agentes.** 1998. Relatório final do projeto de pesquisa PIBIC/CNPq. Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

FURTADO JÚNIOR, Miguel Benedito. **XML**. Rio de Janeiro, 2003. Disponível em: <http://www.gta.ufrj.br/grad/00_1/miguel/index.html>. Acesso em: 01 abr. 2005.

GNUTELLA. **what is gnutella.** [S.I.], 2000. Disponível em: <http://www.gnutella.com/news/4210>. Acesso em: 28 out. 2004.

KARJOTH, Gunter; LANGE, Danny B.; OSHIMA, Mitsuru. **A security model for aglets. IEEE Internet,** Jul. 1997. Disponível em: <http://www.moe-lange.com/danny/ agletsecurity.pdf>. Acesso em: 04 abr. 2005.

LANGE, Danny B. OSHIMA Mitsuru. **Programing and deploying Java móbile agentes with aglets.** Massachusetts, nov. 1998.

LANGE, Danny B. **Mobile Objects and Mobile Agents: The Future of Distributed Computing?**. Sunnyvale 1998. Disponível em: <http://www.moe-lange.com/danny/ecoop98.pdf>. Acesso em: 04 abr. 2005.

LANGE, Danny; OSHIMA, Mitsuru. **Mobile Agents with Java: the Aglet API. Worldwide web journal**, 1998a. Disponível em: <http://www.moe-lange.com/danny/wwwj .pdf>. Acesso em: 04 Abr. 2005.

LIESENBERG, Fernando. **Desenvolvimento de agentes móveis para monitoramento de computadores em rede.** 2004. 86 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

LOTAR, Alfredo. **XML para programadores ASP**. Rio de Janeiro : Axcel Books, 2001. 160p.

MARTINSSON, Tobias. **Desenvolvendo Scripts XML e WMI para o microsoft SQL server 2000.** 1. ed. Tradução Ariovaldo Griese. São Paulo: Makron Books, 2002.

MILOJICIC, D. et al. **MASIF – the OMG mobile agent system interoperability facility.** In: Proceedings of Mobile Agents: 2. International Workshop MA'98 (set: 1998). Lecture Notes in Computer Science, vol. 1477. Springer Verlag, 1998. p. 50–67.

NAPSTER. **what is napster.** [S.I.], 2000. Disponível em: <http://www.napster.com/>. Acesso em: 28 out. 2004.

RICARTE, I.L.M. **PooJava: Agentes**. [S.I.], 2000. Disponível em: <http://www.da.fee.unicamp.br/courses/PooJava/agentes>. Acesso em: 28 out. 2004.

ROCHA, Rafael R. da. **Redes Peer-to-Peer para compartilhamento de arquivos**. 2003. 20 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Grupo de Teleinformática e Automação, Universidade Federal do Rio de Janeiro, Rio de Janeiro.

RUBINSTEIN, M.G. **Avaliação do Desempenho de Agentes Móveis no Gerenciamento de Redes**. 2001. 101 f. Tese de doutorado do Programa de Engenharia Elétrica, Universidade Federal do Rio de Janeiro, Rio de Janeiro.

SETI: **The search for extraterrestrial intelligence.** [S.I.], 2000. Disponível em: <http://setiahome.ssl.berkeley.edu>. Acesso em: 03 nov. 2004.

SHIRKY, Clay. **Clay Shirky's writings about the internet.** Brooklyn, [1996]. Disponível em: < http://www.shirky.com/ >. Acesso em: 28 out. 2004.

VOSS JÙNIOR, José. **Protótipo de software para compartilhar informações entre computadores através da tecnologia peer-to-peer (P2P), usando a plataforma JXTA.** 2004. 70 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

YURI, Flávia. O poder do grid: milhões de máquinas juntas têm a força dos supercomputadores. **Info exame**, São Paulo, n. 195, p. 90-91, jun. 2002.