MANUAL DE UTILIZAÇÃO DO AGLETS

AGLETS – AGENTES MÓVEIS EM JAVA

Autor: Galeno Jung

DAS 6607

INTELIGÊNCIA ARTIFICIAL APLICADA
À CONTROLE E AUTOMAÇÃO

DAS – UFSC

Prof. Ricardo J. Rabelo

INDICE

1- INTRODUÇÃO **3**

2 – AGLETS **5**

2.1 – MOBILIDADE DO *AGLET* 6

3 - J-AAPI **6**

3.1 - CICLO DE VIDA DE UM *AGLET* 7

4 – O MODELO DE EVENTOS DO *AGLETS* **8**

5 - ATP - PROTOCOLO DE TRANSFERÊNCIA DE AGENTES **9**

5.1 – MENSAGEM ATP 10

6 – INTERAÇÕES DA PLATAFORMA *AGLETS* **11**

6.1 – INTERAÇÕES ENTRE *AGLETS* 11
6.2 - TROCA DE MENSAGENS ENTRE *AGLETS* 12
6.1 - FILA DE MENSAGENS E PRIORIDADE 13
6.2 - SINCRONIZANDO MENSAGENS 13
6.3 - MENSAGENS REMOTAS NO *AGLETS* 13

7 – INFRAESTRUTURA PARA O DESENVOLVIMENTO DE APLICAÇÕES COM *AGLETS* **14**

7.1 – *AGLET SERVER* (TAHITI) 14
7.2 – AGLET BOX 15
7.3 – HTTP MESSAGE HANDLING 15
7.4 – FIJI 15
7.5 – HTTP TUNNELING 16

8 - SEGURANÇA EM *AGLETS* **16**

8.1 - MODELO DE SEGURANÇA *AGLETS* 16
8.2 – AUTORIZAÇÕES DE SEGURANÇA 17

9 – PADRÕES DE DESENVOLVIMENTO DO *AGLETS* **18**

10 – INSTALAÇÃO E UTILIZAÇÃO DO *AGLETS-2.0.2* **19**

DESCRIÇÃO 19
REFERÊNCIA NO TEXTO 19
SUGESTÃO 19
10.1 – INSTALAÇÃO NO WINDOWS NT, XP E 2000: 19
10.2 - INSTALAÇÃO NO WINDOWS 98 20
10.3 – INSTALAÇÃO NO GNU/LINUX 21
10.4 – UTILIZAÇÃO DO *AGLETS*/TAHITI 22

11 - CONCLUSÃO **22**

12 - REFERÊNCIAS **23**

1- INTRODUÇÃO

A expansão das redes de computadores no mundo dos negócios tem estimulado diversas pesquisas em sistemas distribuídos de larga escala. Em resposta a este sucesso e ao interesse despertado tanto pelas empresas quanto pela academia, novas técnicas, linguagens e paradigmas surgiram para facilitar a criação de aplicações para diversas áreas. Talvez o mais promissor entre os novos paradigmas seja o que incorpora a noção de agentes móveis.

Agentes Móveis são programas que podem migrar, através de uma rede heterogênea, de um computador para o outro. Ao chegar no destino, os agentes devem se identificar para obter acesso a dados e a serviços do computador. Neste novo sítio, o agente móvel é capaz de (re)começar ou continuar sua execução.

Os agentes móveis desempenham um importante papel na área de objetos distribuídos, pois estes oferecem um paradigma fácil e simples para esta área, com ações síncronas* e assíncronas**, tratamento e envio de mensagens e objetos, tanto móveis como estacionários. Além da mobilidade, os agentes móveis têm outras características [2]:

- Quando um agente se move, todo o objeto move-se junto (código, dados, estado de execução, itinerário e histórico);
- O agente móvel tem capacidade de decidir o que e quando fazer, onde e quando ir;
- O agente móvel pode ser executado de maneira assíncrona, isto é, tem sua própria *thread;*
- Para se comunicar com outros agentes, o agente móvel pode mandar **agentes mensageiros** ou se comunicar localmente com eles;
- O agente móvel pode operar com a conexão da rede funcionando ou não. Caso a conexão não esteja funcionando, este pode esperar até que a conexão seja reaberta;
- Pode-se mandar vários agentes móveis para diferentes sítios para operarem paralelamente.

O interesse na tecnologia de Agentes Móveis é crescente. Isto se deve aos benefícios que este novo paradigma trás para a criação de sistemas distribuídos, tais como [1]:

1. **Redução da Carga da Rede:** o grande número de iterações na comunicação de sistemas distribuídos é eliminado empacotando as negociações transferindo-as ao sítio destino para que esta seja realizada localmente.
2. **Diminuição da Latência da Rede:** o controle de sistemas tempo-real gera uma grande latência na rede. A solução utilizando agentes móveis seria disparar um agente para agir localmente e executar diretamente as instruções do controlador.
3. **Encapsulamento de Protocolos:** agentes móveis estão habilitados a mover-se para sítios remotos e estabelecer canis baseados em protocolos proprietários, tornando desnecessária a constante atualização dos protocolos.
4. **Execução Assíncrona e Autônoma:** um agente pode ser enviado a um dispositivo móvel (PDA, celular, etc) para operar de forma assíncrona e autônoma, barateando as tarefas que requerem conexão aberta e contínua para serem executadas.

---

*Ações síncronas: ações que são executadas instantaneamente, interrompendo qualquer linha de execução.
**Ações assíncronas: ações que não são executadas instantaneamente. Elas executam num instante determinado.

5. **Adaptação Dinâmica:** agentes móveis têm a habilidade para sentir seu ambiente e reagir autonomamente às mudanças. Agentes móveis múltiplos podem se distribuir entre os sítios de forma a manter uma configuração ótima para a realização de tarefas.
6. **Naturalmente Heterogêneos:** como os agentes móveis dependem apenas de seu ambiente de execução (e não independentes da camada de transporte e do computador), fornecem condições ótimas para a integração de sistemas.
7. **Robustos e Tolerantes a Falhas:** os agentes móveis têm a habilidade de reagir dinamicamente a situações e eventos desfavoráveis, tornando assim, fácil a construção de sistemas distribuídos com estes.

Porém, ainda há preocupações quanto à robustez (necessidade de serviços extra nas máquinas onde os códigos irão executar) e a segurança dos sistemas baseados em agentes (segurança dos agentes, das plataformas e da rede subjacente), pois a implementação e o gerenciamento destes têm se mostrado uma tarefa complexa. As questões de segurança serão tratadas em detalhes na seção 8.

O paradigma de agentes móveis tem mostrado grande aplicabilidade em várias áreas como na consulta de informação distribuída, em documentos ativos, em serviços de telecomunicações, no controle e configuração de equipamentos, no gerenciamento e na cooperação de atividades do tipo *workflow,* em redes ativas e no comércio eletrônico. Apesar disso, a tecnologia de agentes móveis ainda se encontra imatura devido a problemas de segurança, de desempenho, ao pouco domínio das plataformas existentes, a falta de comprovações experimentais da aplicabilidade da tecnologia e também devido aos problemas de terminologia, por exemplo, os causados pelo uso equivocado do termo **Agentes Móveis.**

Em 2001 foi publicado pela OMG (*Object Management Group*) um documento que trata da interoperabilidade entre sistemas de agentes móveis chamado MAF – *Mobile Agent Facility.* Este documento apresenta um modelo conceitual comum para o paradigma de Agentes Móveis, descrevendo as principais características de um sistema baseado neste paradigma como a definição de um Sistema de Agentes, a Serialização de Agentes, o modo de localização dos Agentes entre outras.

Este texto tem como objetivo introduzir a plataforma de agentes móveis *Aglets* [3], descrevendo suas principais características e funcionalidades. Para isso, haverá uma breve descrição do sistema de agentes *Aglets.* Em seguida será descrita a J-AAPI (*Java – Aglet Aplication Program Interface*) com suas características e classes principais.

Uma importante característica desta plataforma de agentes é seu modelo de programação, que será exposto em detalhes separadamente. Logo após, haverá uma rápida abordagem sobre o protocolo de transferência de agentes (ATP – *Agent Transfer Protocol*) o qual é usado nas operações com *Aglets.* Após a apresentação do ATP serão descritas as iterações entre os agentes, que são feitas através de mensagens. Seguirão também seções sobre as ferramentas existentes baseadas em *Aglets* (como os servidores de *Aglets Tahiti* e *Aglet Box),* sobre a segurança da plataforma *Aglets,* sobre os padrões de desenvolvimento observados na plataforma *Aglets* além de uma descrição passo a passo da instalação do pacote *Aglets*.

2 – AGLETS

Uma nova plataforma de agentes móveis, chamada *Aglets* (união das palavras *applet* e *agent*), foi lançada em 1996, resultado de trabalhos desenvolvidos nos laboratórios da IBM do Japão [3]. *Aglets* são agentes móveis baseados na linguagem de programação Java. Estes são objetos móveis que podem visitar sítios de uma rede de computadores. Uma vez executando em um sítio, um *Aglet* pode parar sua execução repentinamente, ser despachado para um outro sítio e retomar sua execução neste novo sítio, pois o *Aglet* leva consigo seu código e seu estado [1]. Os *Aglets* são executados dentro de um contexto-*aglet*; este é o espaço de trabalho dos *Aglets* e tem como responsabilidade gerenciar os agentes em execução. Recentemente a IBM disponibilizou o código fonte do sistema sob licença pública, aprovada como uma iniciativa de código aberto.

Com relação à confiabilidade, o *Aglets* tem uma característica instantânea que permite aos *aglets* serem verificados (através de primitivas de *checkpoint*) e reativados se estes forem acidentalmente terminados. As primitivas *checkpoint* criam uma representação do estado do agente que pode ser armazenada em uma memória não volátil, para ser retomada quando necessário.

Atualmente, há pelo menos três áreas nas quais o papel da tecnologia *Aglet* se mostra útil:

1. **Como um mecanismo de comunicação:** *Aglets* são despachados de um sítio para outro através do protocolo ATP que é baseado em URLs. Para tal ação, a URL deve ter a seguinte sintaxe: *atp://<sítio>:<porta>/<contexto>*. O manipulador ATP deve estar devidamente configurado para o ambiente Java para que este ambiente faça o transporte do *Aglet.* Também pode haver casos em que duas aplicações precisam trocar mensagens. Se estas aplicações são *Aglets,* estas podem trocar mensagens utilizando os recursos que o ASDK fornece para comunicação entre agentes. Por outro lado, se os componentes não são *Aglets,* eles podem instanciar *Aglets* para fazer a troca de mensagens por eles.
2. **Como um veículo de transporte de dados através de sítios:** quando é necessário o transporte de algum tipo de dado de um local para outro, um *Aglets* pode transportar este dado como uma variável própria.
3. **Como uma estrutura de divisão das funcionalidades da aplicação:** aplicações deste tipo, que têm como objetivo a divisão das funcionalidades em poucas e conhecidas máquinas (estrutura estática) são desenvolvidas mais facilmente em Java. Porém, quando o número de máquinas é grande, não totalmente conhecido ou variável (estrutura dinâmica), *Aglets* são bons candidatos a resolver este problema.

Como foi dito anteriormente, a linguagem de programação escolhida para o desenvolvimento da plataforma *Aglets* foi Java. Esta linguagem tem várias características que a tornam bastante atraente para o desenvolvimento de sistemas de agentes, tais como [5]:

- Portabilidade do código compilado (arquivos .class);
- Transferência sob-demanda;
- Controle de segurança configurável;
- Suporte para programação de redes em baixo nível;
- Biblioteca completa de classes e interfaces relacionadas com URLs.

Devido às boas características da plataforma Java citadas acima, há vários outros sistemas de agentes móveis baseados nesta linguagem, tais quais: Grasshopper, Concordia, Voyager, Mole, MOA, SOMA e Ajanta.

2.1 – Mobilidade do *Aglet*

Mover um *Aglet* significa tirá-lo de seu contexto atual e transportá-lo para outro contexto. Para esta operação há dois caminhos:

1. **Despachar o *Aglet:*** invocar o método *dispatch* e enviar o agente para uma localização remota, ou melhor, requisitar para que este vá para outro lugar para continuar sua execução.
2. **Retornar o *Aglet:*** invocar o método *retract* e requisitar ao agente que retorne de seu local atual para o seu contexto de origem.

A migração dos agentes no *Aglets* é absoluta, uma vez que este requer a especificação da URL dependente da localização do sistema de agentes destino e suporta a abstração de itinerário. A mobilidade fraca é implementada usando serialização de objetos Java, logo, o estado de execução em nível de *thread* não é totalmente capturado.

Quando um *Aglet* é enviado ou retorna de algum contexto, é desativado ou clonado, ele é transformado numa seqüência de bytes e para poder ser transportado. Então, posteriormente ele é refeito e volta ao seu estado normal. Para isso, é necessário implementa a interface `Java.io.Serializable.`

No momento da serialização pode haver certos acontecimentos que devem ser esclarecidos:

- Objetos que são compartilhados por vários agentes são copiados e serializados por valor. Assim, o agente serializado terá a "sua" cópia deste objeto e não o compartilhará mais com outros agentes.
- Como variáveis de classe não fazem parte de nenhum objeto, não são serializadas, sendo assim, perdidas.

Um *Aglet* pode invocar métodos de objetos remotos através do Java RMI. Assim, quando um *Aglet* é enviado a outro contexto com um Objeto remoto, este é automaticamente enviado pelo seu servidor para o contexto destino do agente. Porém, como a plataforma *Aglets* não pode gerenciar servidores de objetos, sempre é criada uma cópia uma outra cópia do objeto no sítio destino.

3 - J-AAPI

Em resposta à necessidade de uma plataforma uniforme para agentes móveis em ambientes heterogêneos, como a Internet, o laboratório de Tókio da IBM desenvolveu a Java *Aglet* API ou J-AAPI. Por ser escrita em Java, a J-AAPI possui uma grande vantagem: uma vez escrito um *Aglet,* este poderá se executar em qualquer máquina que

tiver suporte à J-AAPI sem a necessidade de verificar qual é o sistema operacional ou *hardware* existente na máquina na qual o *Aglet* está sendo executado. [1]

Há algumas abstrações que devem ser esclarecidas para o total entendimento da J-AAPI e suas funções, a saber:

- O **contexto** do *Aglet* é seu habitat, onde o agente vive e de onde ele obtém informações sobre seu ambiente atual. Também é usado pelo agente para enviar mensagens à outros *Aglets* e à ambientes ativos naquele momento. O contexto é também um objeto estacionário que monitora os *Aglets* e mantém o ambiente seguro contra *Aglets* maliciosos.
- O ***proxy*** é um representante do *Aglet.* Este protege um *Aglet* do acesso direto de outros *Aglets* aos seus métodos públicos além de poder dar a localização do *Aglet.* É através do proxy que se manipula o agente e se realizam verificações de segurança do *Aglet.*
- **Mensagem** é um objeto passado entre *Aglets* de forma síncrona ou assíncrona, usado para a colaboração entre *Aglets.*
- O **itinerário** é o caminho que o *Aglet* tem que percorrer.
- O **identificador** é a identidade do *Aglet.* Este identificador é único em toda a vida do *Aglet.*

A J-AAPI é um padrão para programação e criação de *Aglets.* A seguir têm-se as principais classes e interfaces da J-AAPI e suas características:

- **aglet.Aglet:** classe abstrata que define os métodos fundamentais para um *Aglet* controlar seu ciclo de vida e sua mobilidade; todos os *Aglets* devem ser extensões desta classe.
- **aglet.AgletContext:** classe chave da J-AAPI usada para o *Aglet* ter acesso as informações do seu contexto como o endereço do contexto e a lista dos A*glets* deste contexto (ambiente no qual o *Aglet* passa a maior parte de sua vida).
- **aglet.AgletProxy:** interface que serve para um *Aglet* ter acesso a métodos públicos de outro *Aglet* que não podem ser acessados diretamente. Além disso, esta interface proporciona ao *Aglet* a transparência de localização.
- **aglet.Message**: classe que gerencia as mensagens do *Aglet.*
- **aglet.FutureReplay:** interface que determina se a resposta (assíncrona) a uma pergunta anterior está disponível e se pode esperar pelo resultado com um valor de intervalo especificado de forma que possa continuar sua execução se a resposta não for dada dentro de um tempo especificado.

3.1 - Ciclo de Vida de um *Aglet*

Durante seu ciclo de vida, um *Aglet* está sujeito a muitos eventos. Cada um destes eventos, ilustrados na Figura 1, está ligado a um método da classe **Aglet** da J-AAPI, a saber:

- **criação (*creation*):** o *Aglet* toma seu lugar em um contexto, ganha seu identificador e é inicializado. Então o agente está pronto para ser executado e desempenhar a sua função.

- **clonagem (*cloning*):** um *Aglet* nasce exatamente igual ao *Aglet*-pai, porém, o identificador do *Aglet*-filho é diferente. Além disso, o estado do *Aglet*-pai não é copiado no ato da clonagem.
- **envio (*dispatching*)**: é a ação de despachar um *Aglet* de um contexto e inseri-lo em outro, onde este reiniciará sua execução.
- **retorno (*retraction*):** um *Aglet,* previamente despachado, é trazido de volta a um contexto e seu estado volta junto.
- **desativação (*deactivation*):** um *Aglet* é retirado temporariamente de seu contexto e é colocado em um contexto secundário (o *Aglet* é "congelado", seu estado permanece).
- **ativação (*activation*):** um *Aglet* desativado pode ser reativado, voltando a executar a partir do seu estado no momento da desativação.
- **eliminação (*disposal*):** o *Aglet* é retirado de seu contexto e morre. Seu estado é perdido para sempre.

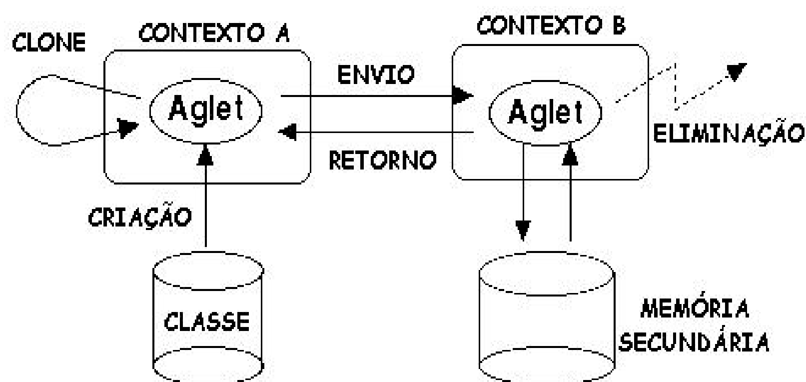

Figura 1 – Ciclo de vida de um *Aglet*

4 – O MODELO DE EVENTOS DO *AGLETS*

Uma característica importante desta plataforma é seu modelo de programação baseado em eventos *callback.* O modelo de programação da plataforma *Aglets* é baseado em eventos, ou seja, o programador constrói tratadores de eventos chamados de *listeners* no código do *Aglet.* Estes *listeners* estarão associados a eventos específicos que acontecem no ciclo de vida do agente. Assim, o programador tem a possibilidade de ligar ações a estes eventos para criar o agente com a função desejada.

Por exemplo, quando um agente chega ao contexto-*aglet* destino, os métodos *onCreation* e *onArrival* são automaticamente invocados. O programador implementa uma classe de agente, herdando implementações padrão desses métodos *callback* da classe **Aglet** e implementa-os com o código específico da aplicação [8]. Tecnicamente, não há limite para o número de *listeners* que pode ser construído em um *Aglet.* Assim, tarefas complexas podem ser divididas em vários *listeners.*

Há três tipos de *listeners*:

- ***CloneListener*:** gerencia os eventos associados à clonagem. Este *listener* responde a três métodos: *onCloning* (logo antes da clonagem), *onClone* (durante a clonagem) e *onCloned* (depois da clonagem).
- ***MobilityListener*:** gerencia os eventos associados à mobilidade do *Aglet.* Este *listener* responde a três métodos: *onDispatching* (logo antes do

despacho), *onReverting* (logo antes da retratação) e *onArrival* (assim que o *Aglet* chega ao destino).

- ***PersistenceListener*:** gerencia os eventos associados à persistência do *Aglet* no contexto. Este *listener* responde a estes dois métodos: *onDeactivating* (logo antes da desativação) e *onActivation* (logo depois da ativação).

5 - ATP - PROTOCOLO DE TRANSFERÊNCIA DE AGENTES

O ATP (*Agent Transfer Protocol*) é um protocolo de transferência de agentes simples e independente de plataforma para transferência de agentes entre computadores ligados em rede. Apesar de existirem várias tecnologias para a programação de agentes (com suas respectivas máquinas virtuais e bibliotecas), o ATP nos dá a oportunidade de manipular agentes móveis de uma maneira geral e unificada:

- Uma máquina que está hospedando agentes possui um serviço de agentes baseado em ATP, que é um componente capaz de receber e enviar agentes, a partir de sítios remotos, via o protocolo ATP. O serviço de agentes é identificado através de um endereço único e independente de plataformas de agentes específicas suportadas pela máquina. Uma máquina pode rodar diversos serviços de agentes.
- Uma máquina pode hospedar diferentes tipos de agentes, desde que ofereça suporte às plataformas de agentes correspondentes.
- Qualquer plataforma de agente deveria incluir um tratador de mensagens ATP.
- Uma mensagem ATP carrega informação suficiente para identificar a plataforma de agente específica (no sítio receptor) e para invocar o tratador ATP que irá atendê-la.

O protocolo ATP é baseado no paradigma de pedido/resposta *(request/reply)* entre máquinas. A máquina A estabelece conexão com a máquina B, envia um pedido para B e espera a resposta. Além disso, todas as ações descritas anteriormente na classe **Aglet** são possíveis através do protocolo ATP [1]. Os métodos descritos abaixo — e ilustrados pela Figura 2 — são suportados pelo ATP:

1. ***Dispatch***: o método *Dispatch* solicita a um sistema de agentes destino que reconstrua um agente a partir do conteúdo de uma requisição e inicie sua execução. Se a requisição obtiver sucesso, o emissor (aquele que disparou o método *Dispatch*) deve terminar o agente e liberar qualquer recurso consumido por ele.
2. ***Retract***: o método *Retract*, solicita a um sistema de agentes destino que envie de volta um determinado agente ao emissor. O sistema que receberá o agente é responsável pela reconstrução do agente. Se a operação for bem sucedida, o sistema que recebeu o pedido de *retract* é quem deve terminar o agente e liberar qualquer recurso consumido por ele.
3. ***Fetch***: como o método *GET* do http, este método solicita a um receptor que este recupere e envie alguma informação (normalmente arquivos.class).
4. ***Message***: o método *Message* é usado para transmitir uma mensagem a um agente identificado por um *agent-id* e retornar alguma resposta à ação. Apesar do ATP adotar a forma de pedido/resposta, não impõe qualquer regra para um esquema de comunicação entre agentes.

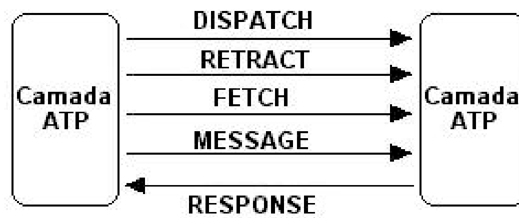

Figura 2 – Operações do ATP

5.1 – Mensagem ATP

O formato de uma mensagem ATP é do tipo:

ATP_message = Request | Response
Message_body = Octet*

Mensagens ATP são constituídas de pedidos e respostas. Todos os pedidos e respostas são compostos por campos de cabeçalho e pelo corpo da mensagem. Um pedido ou request possui o seguinte formato:

Request = Request_Line
Header_Field
[Message_Body]

O campo *Request_Line* indica o método solicitado. Seu formato apresenta a seguinte sintaxe: *Request_Line = Method Resource_URI ATP_version*

onde:

- ***Method***: especifica o método a ser realizado em um recurso que pode ser:
  *Method=**DISPATCH** | **RETRACT** | **FETCH** | **MESSAGE** | Extension_Method*
- ***Resource_URI***: identifica através do formato de uma URI o recurso sobre o qual será aplicado o pedido;
- ***ATP_version***: é um esquema de numeração no formato "*maior.menor*" para indicar versões do protocolo, exemplo ATP/1.2, sendo que estes valores são alterados de acordo com as mudanças do protocolo e suas implicações.

Os *Header_Fields* ou Campos de cabeçalho são associados às mensagens ATP. Alguns estão relacionados somente ao emissor, outros somente ao receptor e alguns estão relacionados a ambos. De acordo com a especificação do ATP, os seguintes campos são usados com os quatro métodos padrão (DISPATCH, RETRACT, FETCH e MESSAGE):
Date, User-Agent, From, Agent-System, Agent-Language, Content-Type, Content-Encoding, Content-Length e *Agent-Id.*

Quanto ao *Message_body*, seu formato e sistema de codificação são definidos pelos campos *Content-Type* e *Content-Encoding* respectivamente, o tamanho do corpo da mensagem está especificado no campo *Content-Length.*

A resposta (response) apresenta o seguinte formato:

Response = Status_Line

Header_Field
[Message_Body]

- O campo *Status_Line* possui a seguinte sintaxe:
  Status_Line = ATP_version Status_code Reason_phrase

onde:
- ***Status_code***: código de três (3) dígitos resultado de uma tentativa do receptor de processar a mensagem. Pode indicar sucesso, ações adicionais necessárias, recurso não presente ou inexistente, erro no pedido, proibições entre outros;
- ***Reason_phrase***: descrição textual do código de status (*status_code*).

6 – INTERAÇÕES DA PLATAFORMA *AGLETS*

6.1 – Interações entre *Aglets*

Para interagirem entre si, os *Aglets* normalmente não invocam os métodos dos outros agentes diretamente. Em vez disso, eles utilizam objetos `AgletProxy`, que agem como representantes do agente. Esta relação entre agente e seu *proxy* é ilustrada na Figura 3.

A classe `AgletProxy` contém métodos que fornecem aos *Aglets* maneiras para solicitar ações de outros *Aglets,* como os métodos `dispatch()`, `clone()`, `deactivate()` e `dispose()`. O agente que recebe uma solicitação para agir pode aceitar esta solicitação, rejeitá-la ou agir mais tarde.

Um *Aglet* tem que utilizar um objeto *proxy* para interagir com outro Ag*let,* mesmo que ambos estejam no mesmo contexto. Os *Aglets* não têm a permissão para interagir com outros *Aglets* diretamente, pois seus métodos de inicialização e *callback* são públicos. Estes métodos deveriam ser invocados apenas pelos contextos-*aglet,* porém, se um agente puder controlar outro, aquele poderá invocar os métodos *callback* e de inicialização do *Aglet* controlado. Um agente se tornaria muito confuso se outro pudesse, maliciosamente ou inadvertidamente, invocar seus métodos de forma direta.

O agente representado pelo seu *proxy* pode ser local ou remoto, mas o *proxy* é sempre local. Por exemplo, se um `BossAglet` em Brusque quiser se comunicar com um `EmployeeAglet` em Florianópolis, o `BossAglet` obtém um objeto *proxy* local que, representa o `EmployeeAglet` remoto. O `BossAglet` simplesmente invoca métodos no *proxy* local, que estabelece uma comunicação através da rede com o `EmployeeAglet`. Apenas *Aglets,* não *proxies,* migram pela rede. Um *proxy* se comunica com um agente remoto que ele representa enviando dados através da rede.

Para obter um *proxy* há três caminhos, sendo que cada um envolve invocar um método no objeto contexto:

1. Criando o *Aglet* com o método `createAglet()`. Esta ação retorna um objeto *proxy* do novo *Aglet.*

2. Procurando numa listagem de *proxies* locais retornada pelo método `getAgletProxies()`.
3. Fornecendo um "identificador *Aglet*" e, se remoto, a localização do *Aglet* como parâmetros do método `getAgletProxies()`.

6.2 - Troca de mensagens entre *Aglets*

A comunicação entre aglets ou entre aplicações e *Aglets* é feita através da troca de mensagens. Para realizar a comunicação é criado um objeto mensagem, com um determinado tipo e um objeto argumento opcional. No destinatário, o objeto mensagem passa por um tratador que verifica qual o seu tipo e obtém os parâmetros que estão no objeto argumento. O tipo indica qual a ação que deverá ser realizada pelo tratador.

Quatro tipos de troca de mensagens são suportados pelo *Aglets,* todos estes são invocados através do `AgletProxy`, como ilustra a figura 3:

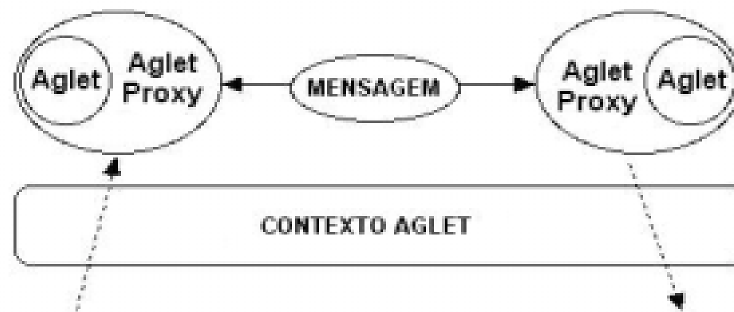

Figura 3 – Relação entre *Aglet* e seu *Proxy*

`Now-type`: `AgletProxy.sendMessage(Message msg)`

Uma mensagem *Now-type* é síncrona, ou seja, a execução atual é bloqueada até que seu receptor tenha concluído o seu tratamento.

`Future-type`: `AgletProxy.sendAsyncMessage(Message msg)`

Esta mensagem é assíncrona, então, não bloqueia a execução atual. Um objeto do tipo *FutureReply* é retornado e pode ser usado para obter o resultado num momento posterior. Quando um aglet envia para si mesmo uma mensagem deste tipo, esta mensagem não é encaminhada para o final da fila de espera, e sim na cabeça da fila e executada imediatamente.

`Oneway-type`: `AgletProxy.sendOnewayMessage(Message msg)`

É uma mensagem assíncrona. Sua diferença em relação à última, está no fato de que é posicionada ao final da fila de mensagens, quando o agente enviá-la para si mesmo.

`Delegation-type`: `AgletProxy.delegateMessage(Message msg)`

Passa um argumento que pode ser usado para realizar a operação de acordo com a mensagem.

Também há maneiras para enviar respostas mesmo sem ter concluído o tratamento da mensagem, o que pode ser feito através do método `Message.sendReply()`.

6.1 - Fila de Mensagens e Prioridade

Todos os objetos *Aglet* têm um objeto fila, onde são armazenadas todas as mensagens recebidas. As mensagens são recebidas de acordo com sua ordem de envio, ou seja, se o emissor enviar três mensagens na ordem "A", "B", "C", o receptor as receberá na mesma ordem. Prioridades podem ser atribuídas às mensagens, de maneira que mensagens com prioridades de valores mais altos são atendidas primeiramente. A prioridade pode ser atribuída através do método `MessageManager.setPriority(String, int)`. Há um tipo de prioridade chamado de prioridade *NOT_QUEUED*, que faz com que a mensagem não seja enviada para a fila, mas sim transmitida imediatamente ao aglet e atendida em paralelo.

6.2 - Sincronizando Mensagens

Embora as mensagens sejam tratadas uma por vez, é comum existirem casos de dependência de que certas condições sejam satisfeitas para o tratamento da mensagem prosseguir. No Aglets há métodos que auxiliam o suporte a este tipo de problema, são cinco métodos para a sincronização de mensagens:

`Aglet.waitMessage()`: faz com que a *thread* atual espere até que outra mensagem ordene-a a retomar a sua execução. Se não receber a ordem para retomar a execução, irá esperar indefinidamente, até que seja interrompida.

`Aglet.waitMessage(long timeout)`: faz com que a *thread* aguarde por no máximo *timeout* milisegundos e acorde quando expirar o tempo especificado.

`Aglet.notifyMessage()`: ordena a *thread* em espera a acordar e retomar a execução.

`Aglet.notifyAllMessages()`: ordena a todas as *threads* em espera, que retomem suas execuções.

`Aglet.exitMonitor()`: ordena a *thread* da execução atual a liberar o monitor que está possuindo.

6.3 - Mensagens Remotas no *Aglets*

Aglets oferece suporte à troca de mensagens de maneira remota, permitindo que objetos aglets comuniquem-se de forma remota assim como localmente. Os argumentos ou valores de retorno transmitidos através de troca de mensagens remota podem ser de qualquer tipo *Java* que implemente a classe `java.io.Serializable`.
A diferença entre a troca de mensagens remota e o envio de um aglet é que não há a transferência de *bytecodes* na troca remota, assim, é necessário que as classes usadas na mensagem, estejam instaladas nos dois lados envolvidos. As vantagens do uso de troca remota estão no fato de que esta é uma forma de comunicação mais leve entre dois aglets que estão em hosts diferentes, também estão na redução do tráfego da rede, do custo de definir classes e melhora de questões relacionadas à

7 – INFRAESTRUTURA PARA O DESENVOLVIMENTO DE APLICAÇÕES COM *AGLETS*

Há várias ferramentas baseadas em *Aglets* que facilitam o entendimento deste sistema de agentes. Estes programas auxiliam o usuário de várias formas, tais quais: no aprendizado da plataforma *Aglets,* no desenvolvimento de aplicações com *Aglets,* no monitoramento de *Aglets,* nos testes iniciais. A seguir, haverá uma descrição de algumas das principais ferramentas desenvolvidas para esta plataforma de agentes.

7.1 – *Aglet Server* (Tahiti)

Esta ferramenta serve para gerenciar os *Aglets* criados pelos usuários. O Tahiti *Aglets Server* é composto por uma combinação de um *Aglet server standalone* com um ATP (*Agent Transfer Protocol*) *daemon,* um contexto e um vizualizador [2].

A Figura 4 mostra o Tahiti *Aglets Server.* Esta ferramenta tem uma interface gráfica que fornece várias ações possíveis de serem feitas com *Aglets.* Também é possível mudar configurações de segurança, do ambiente, de rede entre outras facilidades que o Tahiti dispõe para o usuário.

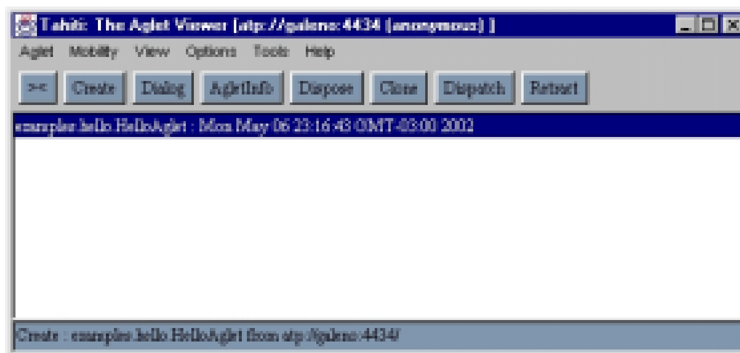

Figura 4 – *Aglet Server* (Tahiti)

Através da Figura 5 pode-se observar a disposição dos *menus* do Tahiti, que se dividem em:

- ***Aglet:*** opções de gerenciamento de *Aglets.* Através das funções deste *menu* pode-se criar um *Aglet*, clonar um *Aglet*, obter informações de qualquer *Aglet* residente no contexto, eliminar um *Aglet* entre outras ações.
- ***Mobility:*** opções de Mobilidade dos *Aglets.* As funções presentes neste *menu* permitem ao usuário que envie um *Aglet* a um contexto remoto, que solicite o retorno de um *Aglet* além de possibilitar a desativação e ativação de agentes residentes no contexto.
- ***View:*** *menu* que fornece o histórico (*log records*) e a memória usada pelos *Aglets* residentes neste.
- ***Tools:*** *menu* que fornece ferramentas para o trabalho com *Aglets,* como a *Garbage Collection,* visualização da *thread* atual entre outer funções.
- ***Help:*** fornece informações sobre o Tahiti e a plataforma *Aglets* além de links para sítios na Internet onde o usuário pode tirar dúvidas e encontrar mais textos sobre *Aglets.*

Além dos *menus* citados acima, o Tahiti *Aglet Server* fornece o *menu Options* onde o usuário tem a possibilidade de mudar várias configurações do ambiente. O *menu Options* está subdividido em quatro seções:

1. ***General Preference:*** contém configurações do ambiente. Há a possibilidade de: configurar a fonte usada pelo Tahiti, escolher um *Aglet* para ser inicializado junto com o Tahiti, configurar a ordem de visualização dos *Aglets,* vizualizar o *Cachê* além de mudar as informações do usuário.
2. ***Network Preference:*** possibilita a configuração de opções de rede do Tahiti, como: habilitar o HTTP Tunneling, habilitar a aceitação de pedidos (*requests*) HTTP como mensagens além das configurações de segurança para *login* do Tahiti.
3. ***Security Preference:*** fornece opções de segurança do Tahiti. Possibilita a configuração do privilégio de acesso de vários itens (*Aglet, Window, Context,* etc).
4. ***Sever Preference:*** possibilita a especificação de um diretório através do qual o Tahiti obterá os *Aglets* inseridos pelo usuário.

Figura 5 – Menus do Tahiti

7.2 – Aglet Box

Servidor simples implementado em Java que serve para armazenar despachados para servidores remotos momentaneamente indisponíveis. Os *Aglets* ficam numa *Aglet Box* privada até que o servidor para o qual foram despachados esteja disponível. [2]

7.3 – HTTP Message Handling

Ferramenta que serve para enviar diretamente aos *Aglets* mensagens em HTTP (*HiperText Transfer Protocol*), como requisições HTTP/CGI de um browser. [2]

7.4 – Fiji

Plugin (aditivo para softwares) mais uma biblioteca capaz de criar *applets* que rodam contextos *Aglet* e podem criar e despachar *Aglets* [2]. Porém, o desenvolvimento e a distribuição deste *plugin* estão suspensos.

7.5 – HTTP Tunneling

É uma maneira (suportada pelo ATP) para encapsular no corpo de mensagens em HTTP, qualquer um dos métodos do ATP, possibilitando assim o **envio** e o **retorno** de *Aglets* através de *firewalls*. Isto só é possível se os servidores envolvidos estiverem adequadamente configurados para receber pedidos em ATP "empacotados" em uma mensagem HTTP. Porém, por limitações impostas pelos *firewalls,* o *Aglet* despachado não pode buscar uma classe sob demanda através de um *firewall* nem se autodespachar através deste tipo de barreira.

8 - SEGURANÇA EM *AGLETS*

(Este capítulo está incompleto)

Há uma grande preocupação na questão de segurança de sistemas de agentes móveis, pois, em um ambiente de larga escala nem o agente móvel nem as máquinas onde este é executado podem ser considerados confiáveis. Estas preocupações são: a proteção das plataformas contra agentes maliciosos, a proteção dos agentes contra outros agentes, a proteção dos agentes contra plataformas maliciosas e a proteção da rede de computadores subjacente.

Como foi dito anteriormente, um *Aglet* é um objeto em Java que migra de um *host* para outro. Sendo assim, este pode acessar todos os recursos do sítio que o hospeda. Como o *Aglet* está baseado no Java, o agente se utiliza (e confia) na segurança do interpretador Java, ou seja, a segurança do *Aglet* depende da segurança do interpretador Java que está no *host* hospedeiro.

Para a segurança dos *Aglets,* há também o **Modelo de Segurança *Aglets*** [6] que define quem e sob quais condições os objetos de um contexto poderão ser acessados, qual o tipo de autenticação necessária para permitir o acesso, a segurança da comunicação entre *Aglets* e os níveis de segurança possíveis.

8.1 - Modelo de Segurança *Aglets*

Os Integrantes do laboratório de pesquisa da IBM do Japão apresentam um modelo de segurança para a plataforma *Aglets* em (Karjoth et al. 1997), porém este modelo está apenas parcialmente implementado na verão corrente do ASDK, que suporta as seguintes funcionalidades de segurança:

- Autenticação de usuários de um domínio;
- Integridade da comunicação entre plataformas de um mesmo domínio;
- Autorização com granularidade fina similar ao modelo de segurança do JDK1.2.

Um domínio no *Aglets* corresponde a um conjunto de contextos (lugares na terminologia da OMG) que seguem as mesmas políticas...

TEXTO DA MICHELE...

8.2 – Autorizações de Segurança

Quando um *Aglet* pretende acessar recursos (como propriedades Java, *threads,* etc), este acesso deve ser controlado através das permissões concedidas àquele agente. Estas permissões podem ser especificadas a partir do Tahiti ou diretamente, editando as políticas de segurança.O formato das políticas de segurança usadas na plataforma *Aglets* foi desenvolvido para estar de acordo com o JDK1.2 (Java2). O programador de *Aglets* está apto a especificar as seguintes permissões na política de segurança:

1. **Propriedades Java**

`java.io.FilePermission` : ler/modificar/executar arquivos
`java.net.SocketPermission` : abrir/aceitar/resolver/conectar *Sockets*
`java.awt.AWTPermission` : permissões de janelas
`java.util.PropertyPermission` : propriedades Java
`java.lang.RuntimePermission` : propriedades em tempo de execução
`java.security.SecurityPemission` : propriedades das políticas de segurança
`java.security.AllPermission` : outras permissões Java

2. **Propriedades *Aglets***

`com.ibm.aglets.security.ContextPermission` : propriedades do contexto
`com.ibm.aglets.security.AgletPermission` : envio, clonagem, etc
`com.ibm.aglets.security.MessagePermission` : troca de mensagens

Para especificar diretamente as permissões, o programador deve dizer qual *codebase* seguir o seguinte modelo:

```
grant codebase "codebase","ownedBy"

{
      permission name.of.the.permission "toWho","kindOfPermission";
}
```

Por exemplo, se Galeno quer que um *codebase* seu, chamado `"anycodebase"`, tenha direito de leitura ao diretório "c:\temp" e direito de eliminar qualquer *Aglet* que pertence à Galeno, ele deve escrever seu código da seguinte maneira:

```
grant codebase "anycodebase","Galeno"
{
      permission java.io.FilePermission "c:\temp","read";
      permission com.ibm.aglets.security.AgletPermission "Galeno","dispose";
}
```

Por outro lado, estas mesmas permissões poderiam ser especificadas no Tahiti, na opção *Security Preferences* do menu *Options.* A figura X mostra esta opção.

FIGURA X

O lado esquerdo da figura (campo *Codebase*) indica o nome do *codebase* (`Codebase`), a quem ele pertence `(Owned By)` e ... `(Signed By)`. O lado direito da figura X indica as permissões relacionadas ao *codebase* selecionado no lado esquerdo. O menu na parte superior direita da figura indica o tipo de permissão selecionada (no caso XXXX) e o campo abaixo mostra as permissões associadas ao *codebase* selecionado.

9 – PADRÕES DE DESENVOLVIMENTO DO *AGLETS*

O conceito de padrões de desenvolvimento surgiu na comunidade de engenharia de software da orientação a objetos. Hoje é reconhecido como uma das mais significantes inovações neste campo.

Padrões de desenvolvimento (*Pattern Design*) são métodos de programação que podem ajudar na criação de agentes, "capturando" boas soluções para problemas que aparecem freqüentemente no projeto de agentes. Estes padrões auxiliam os desenvolvedores de aplicações baseadas em agentes no sentido de torná-las mais flexíveis, reutilisáveis e mais fáceis de entender (motivos que fizeram muitos se interessarem pela tecnologia de Agentes Móveis).

Em [7], os autores sugerem um esquema para classificação dos padrões de desenvolvimento de agentes (e descrevem alguns destes padrões) identificados na plataforma *Aglets.* Segundo este esquema, três classes de padrões podem ser identificadas:

1. ***Traveling:*** esta classe de padrões é a essência dos Agentes Móveis, pois lida com vários aspectos do gerenciamento da mobilidade dos agentes, como rotas, itinerários e qualidade de serviço. Estes padrões possibilitam o encapsulamento da gerência de mobilidade, o que simplifica o desenvolvimento dos agentes. Segue abaixo uma tabela com os três padrões do tipo *traveling* existentes na plataforma *Aglets:*

Itinerary: mantém uma lista de destinos e define a rota (o itinerário) além de gerenciar casos especiais (como quando um destino não existe).
Forwarding: maneira automática de enviar a outro sítio *Aglets* recém chegados em um sítio.
Ticket: através de um *ticket*, um agente pode solicitar a qualidade de serviço e as permissões necessárias para ir a outro sítio e assegurar sua execução.

2. **Task:** classe de padrões para a análise e controle das tarefas dos *Aglets.* Esta classe possibilita a divisão de tarefas entre *Aglets* e a cooperação entre estes *Aglets,* por exemplo. A tabela abaixo mostra os dois padrões do tipo *Task* da plataforma *Aglets:*

*Master-Slave**:*** define um esquema através do qual um agente mestre delega ações a um agente escravo que viaja até o destino, executa sua tarefa e volta a origem para apresentar os resultados ao agente mestre.
Plan: fornece uma maneira de monitoramento de tarefas multi-agentes, como ações com vários agentes trabalhando em paralelo, por exemplo. Esconde dos agentes o fluxo de tarefas e estes fornecem sua mobilidade para a realização da tarefa.

3. ***Interaction:*** esta classe de padrões foi implementada no sentido de facilitar a interação entre *Aglets.* Abaixo segue uma tabela com os padrões do tipo *Interaction* da plataforma *Aglets:*

Meeting: fornece uma maneira para dois ou mais *Aglets* iniciarem interação local num sítio dado. Os agentes vão ao local de encontro (*meeting place*) onde são avisados da chegada dos outros agentes para iniciar a iteração.

Locker: define um compartimento privado de memória para armazenar dados de um *Aglet* que foi temporariamente enviado a outro sítio. Assim, este agente pode voltar posteriormente e reaver os dados armazenados.
Messenger: define um agente mensageiro para carregar uma mensagem a um agente localizado em outro sítio.
Facilitator: define um agente que fornece serviços para nomeação e localização de agentes, inclusive dando nomes simbólicos aos agentes. Este agente se movimentaria pelos sítios recebendo as atualizações de localização dos outros agentes.
Organized Group: compõe grupos de agentes com os quais todos os membros de um grupo viajam juntos.

Os padrões de desenvolvimento de agentes têm provado sua grande funcionalidade no campo da orientação a objetos. Estes padrões têm facilitado a introdução de leigos no mundo dos agentes móveis, uma vez que facilitam o entendimento e a implementação de agentes nas mais diversas áreas e para as mais variadas aplicações.

10 – INSTALAÇÃO E UTILIZAÇÃO DO *AGLETS-2.0.2*

A tabela abaixo mostra os itens necessários, e seus respectivos endereços, para instalação do pacote aglets-2.0.2 com JDK1.4 (*Java Development Kit*):

Descrição	**Referência no texto**	**Sugestão**
Pacote Java	*(JDK_HOME)*	C:\jdk1.4
Pacote *Aglets*	*(AGLET_HOME)*	C:\aglets-2.0.2
Pacote **Ant**	*(ANT_HOME)*	C:\ant

Procedimento padrão para qualquer Sistema Operacional:

- Instalar o JDK [4];
- Descompactar o pacote aglets-2.0.2 [3];

10.1 – Instalação no Windows NT, XP e 2000:

- Criar o arquivo **SetPaths.bat** com o seguinte conteúdo:

```
set JAVA_HOME=(JDK_HOME)
set ANT_HOME=(AGLET_HOME)
set JDK_HOME=(JDK_HOME)
set AGLET_HOME=(AGLET_HOME)
set PATH=%PATH%;(JDK_HOME)\bin;(JDK_HOME)\lib;(AGLET_HOME)\lib;
(AGLET_HOME)\bin
set CLASSPATH=(JDK_HOME)\lib;(JDK_HOME)\JRE\lib;(AGLET_HOME)\lib\aglets-2.0b0.jar;
(AGLET_HOME)\public;(AGLET_HOME)\lib;(AGLET_HOME)bin.
```

- Executar o arquivo **SetPaths.bat**;
- Executar o arquivo **ant.bat** localizado no diretório *(AGLET_HOME)*\bin;

- Executar a linha de comando **ant install-home** no diretório *(AGLET_HOME)*\bin para copiar os arquivos .keystore e .java.policy para o diretório C:\Windows;
- Rodar o servidor através do arquivo **agletsd.bat** localizado no diretório *(AGLET_HOME)*\bin.

Ao executar o arquivo **agletsd.bat** o servidor Tahiti requisitará *login* e senha:

- Login: *anonymous;*
- Senha: *aglets.*

Uma alternativa ao arquivo **SetPAths.bat** é adicionar as variáveis de ambiente no arquivo **Autoexec.bat**. Além disto, as variáveis PATH e CLASSPATH devem ser alteradas (ou criadas) de acordo com o que foi especificado acima.

Obs.: Ao executar algum arquivo, poderá aparecer o aviso *Sem espaço de ambiente.* Caso isso aconteça, deve-se executar os arquivos num atalho para o *prompt* do DOS com memória alterada. Isso deve ser feito na janela de propriedades do atalho, no menu *memória.*

10.2 - Instalação no Windows 98

A instalação da versão 2.0.2 do Aglets no Win98 exige alguns ajustes:

- Obter o programa *Ant*, disponível em http://ant.apache.org;
- Instalar o JDK [4];
- Descompactar o pacote aglets-2.0.2 [3];

Então, os seguintes passos devem ser executados:

1. Editar o arquivo config.sys (presente em C:) e adicionar a seguinte linha: **shell=c:\command.com c:\ /p /e:32768** Após isto, deve-se reiniciar o PC.

2. Descompactar o programa *Ant:* deve ser em **C:\ant** ou um nome de pasta com até 6 caracteres (importante!).

3. Editar o arquivo **autoexec.bat** (presente em C:) e adicionar as seguintes linhas:

```
set JAVA_HOME=C:\j2sdk1.4.1_02
set ANT_HOME=C:\ant
set AGLET_HOME=C:\aglets-2.0.2
set JDK_HOME=C:\j2sdk1.4.1_02
set
PATH=%PATH%;%JDK_HOME%\bin;%JDK_HOME%\lib;%AGLET_HOME%\lib;
%AGLET_HOME%\bin;%ANT_HOME%\bin;
set
CLASSPATH=%JDK_HOME%\lib;%JDK_HOME%\JRE\lib;%AGLET_HOME%\lib\
aglets-2.0.2.jar;%AGLET_HOME%\lib
```

Salvar e executar o arquivo **autoexec.bat**, ou reiniciar o PC (recomendado). Estas alterações tornam desnecessário o arquivo **SetPaths.bat**

4. Excluir arquivo **ant.jar** da pasta C:\aglets-2.0.2\lib e os arquivos **ant** e **ant.bat** da pasta C:\aglets-2.0.2\bin.

5. Executar a seguinte seqüência de comandos: ANT e ANT INSTALL_HOME

6. O *Aglets* está pronto para rodar. Execute o comando **agletsd.bat**: Login: *Anonymous* - Senha: *Aglets*

10.3 – Instalação no GNU/Linux

1. Criar o diretório onde o *Aglets* ficará. O arquivo **aglets-2.0.2.jar** deve estar lá. Por exemplo: Aglets-2.0.2

2. Dentro deste diretório, extrair o arquivo **aglets-2.0.2.jar**, executando a seguinte linha de comando: **jar -xvf aglets-2.0.2.jar**

3. Em máquinas (como no laboratório LCMI) que têm o **tcshell** (para descobrir se é tcshell ou bash, digitar: **printenv SHELL**), editar o arquivo **.cshrc**. Neste arquivo, especificar PATH, CLASSPATH e variáveis de ambiente. Por exemplo:

```
setenv AGLET_HOME /home/usuario/Aglets-2.0.2
setenv ANT_HOME /home/usuario/Aglets-2.0.2
setenv JDK_HOME /usr/java/j2sdk1.4.1_01
setenv JAVA_HOME /usr/java/j2sdk1.4.1_01
setenv CLASSPATH ${AGLET_HOME}/lib:${JDK_HOME}/lib:${JDK_HOME}/jre/lib:${AGLET_HOME}/public
setenv PATH ${PATH}:/usr/local/bin/:${AGLET_HOME}/bin:${JDK_HOME}/jre/bin:${ANT_HOME}/bin
```

Obs.: Se o **shell** for o **bash**, editar o arquivo **.bashrc** trocando **"setenv"**, por **"export"**. Salvar este arquivo e executar a seguinte linha de comando: **export .cshrc** (ou então **export .bashrc**).

1. Ir em (*AGLETS_HOME*)/bin e executar o comando **ant.** Caso este comando não funcione, as variáveis de ambiente não estão corretamente configuradas. Se o comando **ant** funcionar, pular para o próximo passo.

2. Ainda em (*AGLETS_HOME*)/bin, executar o comando **ant install-home**.

3. No mesmo diretório, executar o comando **agletsd.bat:** Login: *Anonymous* - Senha: *Aglets*

10.4 – Utilização do *Aglets*/Tahiti

A partir do momento que o Aglets-2.0.2 está instalado, o Tahiti está executando com sua janela principal (Figura 4) aparecendo na tela, é possível criar agentes *Aglets*. Para que a criação de agentes seja feita de maneira correta, deve-se saber algumas características deste sistema:

- O agente deve estar compilado para ser criado no Tahiti (arquivo .class);
- O nome do agente não pode ser o mesmo de seu pacote;
- Usa-se, por conveniência, letras minúsculas para nomes de pacotes.

Além disto, os agentes devem estar em pacotes (diretórios) localizados a partir do diretório ***(AGLET_HOME)*/public/.** Por exemplo, para criar no Tahiti o agente SimpleMaster, localizado em ***(AGLET_HOME)*/public/examples/simplemasterslave/**, deve-se seguir os seguintes passos:

1. Clicar no botão *Create* da janela principal do Tahiti, para abrir a janela de criação de agentes;
2. Nesta nova janela, no campo de texto Aglet Name, digitar: examples.simplemasterslave.SimpleMaster;
3. Ainda nesta mesma janela, clicar no botão Create.

Assim, o agente SimpleMaster foi criado com sucesso. É importante observar que o campo de texto Source URL da janela de criação de agentes só é usado para criação remota de agentes, e ainda não foi estudado.

11 - CONCLUSÃO

A ferramenta para a criação de *Aglets* unifica de maneira simples, várias características dos agentes, tais como [1]:

- possibilidade de trabalhar com objetos móveis ou estacionários;
- envio de objetos, de mensagens e de dados;
- objetos autônomos e passivos;
- capacidade de gerenciamento de processos síncronos e assíncronos;
- operações com o *host* conectado ou desconectado da rede;
- viabilidade de interação com objetos locais ou remotos;
- execuções seqüenciais e/ou paralelas.

Além disso, o pacote *Aglets* fornece o servidor *Tahiti* que facilita muito os testes iniciais e o gerenciamento dos agentes.

12 - REFERÊNCIAS

1. [LAN97] LANGE, Danny B.; OSHIMA, Mitsuru. **Programming Mobile Agents in JavaTM - with the Java Aglet API**. 1997.

2. Relatório final de projeto PIBIC/CNPq, 1997. Alexandre Rodrigues Coelho.

3. IBM Corp.: Página oficial do *Aglets.* http://www.trl.ibm.co.jp/aglets/

4. Sun Microsystems Inc.: Java. http://java.sun.com/

5. Jerry Smith – Distributed Computing with Aglets. 1999

6. Aglets Especification 1.1 Draft

7. Agent Design Patterns: Elements of Agent Application Design. Danny Lange, Yariv Aridor.

8. Karnik 1998 – do texto qualify michelle…