Trabalho Final de Curso

1997/1998

Trabalho n.º 55

Cliente-AS

Implementação de um cliente Web para gestão e monitorização de Aplicações Baseadas em Agentes

(Título antigo: Aplicações Domóticas em WWW)

Documento 2

Professores Responsáveis:

Mestre Alberto Silva
Prof. Doutor José Delgado

Trabalho realizado pelos alunos:

Adílio Soares, n.º 39474
João Furtado, n.º 36805

Outubro de 1998

Agradecimentos

Ao Mestre Alberto Silva pelo acompanhamento, orientação e inúmeras sugestões fornecidas ao longo deste ano lectivo, facilitando este trabalho.

Ao Prof. José Delgado, por nos ter sugerido este trabalho.

Aos colegas que connosco partilharam a sala de trabalho do INESC, pela tolerância na partilha de recursos da sala.

Às nossas famílias e amigos, pelo apoio oferecido.

Sumário

Como objectivo principal, pretende-se o desenvolvimento de uma aplicação cliente, denominada Cliente-AS, que possibilite através de uma interface gráfica a gestão e monitorização de Aplicações Baseadas em Agentes. Além disso, através deste cliente deverá ser possível gerir remotamente servidores AgentSpace, efectuando operações tais como: lançar e configurar agentes, gerir locais de execução, gerir contas de utilizadores, grupos de utilizadores e controlar acessos.

Pretende-se também o desenvolvimento de Aplicações Baseadas em Agentes (ABA), de modo a testar a sua funcionalidade e verificar a sua importância, bem como testar o Cliente-AS como mecanismo de gestão de ABA. Estas aplicações devem servir de elemento comparativo entre alguns dos vários sistemas de suporte de agentes existentes.

De forma a possibilitar a gestão remota de servidores AgentSpace (Servidores-AS) e Aplicações Baseadas em Agentes através do Cliente-AS e de forma a facilitar o acesso a todos os utilizadores que o desejassem, este deve suportar-se nas novas tecnologias de informação como a Internet/Web. Desta forma o único requisito necessário à execução do Cliente-AS deve ser a existência de um cliente Web (*browser*) que suporte JDK1.1 instalado na máquina do utilizador.

O Cliente-AS é desenvolvido em Java usando um *applet* que medeia a comunicação entre o utilizador final e os vários recursos mantidos em um ou mais Servidores-AS. O *applet* é carregado a partir de uma página vulgar da Web e providencia um elevado nível de integração com a Internet/Web, daí resultando a sua importância para o trabalho. Quando o *applet* arranca o utilizador pode ligar-se a um Servidor-AS.

Para o mecanismo de comunicação foi utilizado o Voyager. O Voyager é um ORB (Object Request Broker) "100% Java" integrado de forma adequada com o JDK1.1. A sua utilidade é decisiva já que permite ultrapassar a limitação do *applet* só poder comunicar com a máquina da página a partir da qual foi carregado.

Como resultado deste trabalho apresenta-se o Cliente-AS que possibilita a gestão remota de servidores AgentSpace e de Aplicações Baseadas em Agentes. É possível criar agentes, locais de execução, e lançar agentes para efectuarem as tarefas especificadas. Conseguiu-se a gestão controlada dos recursos através do controle de palavras chave que identificam o perfil do utilizador.

Além do cliente, foi desenvolvida uma ABA denominada de CELIA para teste de todo o sistema na sua globalidade. Ou seja, o teste do Servidor-AS e do Cliente-AS como meio de gestão de ABA e do Servidor-AS. Para além disso serviu para demonstrar a importância das ABA.

Conteúdo

Lista de Figuras

1 Introdução

Nesta secção pretende-se fazer uma introdução ao trabalho desenvolvido. Primeiro faz-se um enquadramento geral sobre os sistemas de agentes móveis e sua utilidade como forma de efectuar computações assíncronas na Internet. A seguir apresenta-se uma breve explicação acerca do trabalho pretendido, a sua importância, aplicação, formas de resolução e a solução adoptada. Finalmente, descreve-se a organização do documento.

1.1 Enquadramento

A Internet é um espaço aberto, amplo e de múltiplas culturas em rápido crescimento. Com esse rápido crescimento, quer da rede, quer do seu número de utilizadores, rapidamente surgiram novas tecnologias/paradigmas aos quais os tradicionais sistemas de informação se tiveram de adaptar.

A Web, um dos serviços principais que a Internet disponibiliza e que lhe garante grande parte do seu enorme sucesso, era originalmente um sistema hipermédia distribuído para navegação e consulta de informação. Porém, não disponha de mecanismos para sistemas de informação como os mecanismos proporcionados pelos sistemas de informação tradicionais, como é o caso de formulários (para introdução de informação), interligação a bases de dados, etc. Assim, devido ao crescimento da Internet e da pressão dos seus utilizadores para a introdução de novos mecanismos para consulta de informação, começaram a desenvolver-se sistemas de informação para a Web cada vez mais complexos, sofisticados e mais distribuídos, os chamados Sistemas de Informação para Web (SIW) ou Web Information Systems (WIS) [SMD97].

A certa altura sentiu-se a necessidade de entidades que pudessem representar os utilizadores e efectuar determinadas tarefas em seu nome possibilitando assim computações assíncronas (e até desconectadas) na Internet. Assim aparece o conceito de agentes móveis de software. Esses agentes são componentes de aplicações, chamadas de Aplicações Baseadas em Agentes, que actuam em nome dos seus donos/utilizadores apresentando um conjunto básico de atributos reconhecidos, tais como: autonomia, persistência e sociabilidade. Podem ainda apresentar características adicionais como: inteligência, aprendizagem ou mobilidade.

Os agentes móveis de software necessitam de uma plataforma onde possam ser executados. Aos sistemas de suporte à execução de agentes chama-se Sistema de Suporte de Agentes (SSA). Esses sistemas são, então, a base para a construção e desenvolvimento de Aplicações Baseadas em Agentes (ABA).

Um SSA existente é o AgentSpace da autoria do Mestre Alberto Silva do INESC. Esse trabalho foi desenvolvido no âmbito da sua tese de Doutoramento em Aplicações Baseadas em Agentes móveis e pretende ser uma plataforma de suporte à execução de agentes de forma a permitir o desenvolvimento de aplicações baseadas em agentes.

1.2 Objectivo

O objectivo principal do trabalho consiste no desenvolvimento de uma aplicação cliente denominada Cliente-AS que permita, de forma simples, controlada e segura, a gestão e monitorização remota de servidores AgentSpace (Servidor-AS), e Aplicações Baseadas em Agentes. Para além das operações básicas de um SSA como a gestão de agentes e locais de execução, o Cliente-AS deverá ainda permitir a gestão de utilizadores, grupos de utilizadores, e suas permissões.

1.3 Importância e Aplicações

A necessidade da aplicação descrita é indissociável da importância das Aplicações Baseadas em Agentes e do SSA AgentSpace como infra-estrutura de suporte dessas aplicações, na medida em que é através da aplicação cliente que se efectua a gestão das ABA do SSA AgentSpace. De modo a averiguar a importância das ABA, procedemos ao estudo de vários SSA existentes e desenvolvemos Aplicações Baseadas em Agentes de forma a verificar as suas características. Outro factor relevante para a existência desta aplicação cliente é a possibilidade gerir Servidores-AS e ABA remotamente.

Uma possível aplicação real do trabalho desenvolvido é a gestão de uma ABA de comércio electrónico de livros baseada na Internet/Web de âmbito aberto. Esta aplicação, denominada CELIA, foi desenvolvida com o objectivo de por um lado verificar a importância do paradigma de agentes móveis e por outro comparar diferentes SSA com o AgentSpace.

1.4 Soluções

De forma a possibilitar a gestão remota de Servidores-AS e Aplicações Baseadas em Agentes através do Cliente-AS e de forma a facilitar o acesso a todos os utilizadores que o desejassem, este deve suportar-se nas novas tecnologias de informação como a Internet/Web. Desta forma o único requisito necessário à execução do Cliente-AS deve ser a existência de um cliente Web (*browser*) que suporte JDK1.1 instalado na máquina do utilizador.

O Cliente-AS deverá ser constituído por um ou mais *applets*. A perfeita integração do(s) *applet(s)* com a Internet/Web possibilita o acesso e utilização da plataforma AgentSpace por quem quer que seja, à escala mundial.

1.5 Organização do Documento

Este documento está organizado em secções e cinco apêndices como se descreve de seguida.

Em **Noções Básicas** definimos conceitos que servem de suporte ao trabalho. É o caso da linguagem Java, Sistemas de Suporte de Agentes, Agentes de Software, e ORB – Object Request Broker.

Em **Técnicas** descreve as técnicas utilizadas para a resolução dos problemas. Refere-se o processo de desenvolvimento e ferramentas utilizadas. Descreve-se a infra-estrutura AgentSpace como suporte ao desenvolvimento do Cliente-AS e das Aplicações Baseadas em Agentes. O estudo de Aplicações Baseadas em Agentes é introduzido como forma de verificar a importância das ABA e do SSA AgentSpace. Finalmente descreve-se a arquitectura do Cliente-AS bem como as técnicas utilizadas no seu desenvolvimento.

A secção **Resultados** valida a importância das ABA bem como do SSA AgentSpace referida na introdução. Refere também os objectivos atingidos, e os que ficaram por atingir. Esses resultados foram em parte obtidos através do uso de uma ABA construída para o efeito.

Em **Conclusões** refere a eficiência dos métodos utilizados e propõe desenvolvimentos futuros.

Em **Referências** enuncia-se a bibliografia utilizada apresentada de acordo com as regras de catalogação.

No **Apêndice A** apresenta-se o manual de utilizador da aplicação Cliente-AS.

No **Apêndice B** apresenta-se as regras de criação de meta classes para agentes e gestores de segurança.

No **Apêndice C e Apêndice D** apresentam-se os resultados das duas versões da ABA CELIA relativos aos testes efectuados sobre semânticas de comunicação e mobilidade, respectivamente.

No **Apêndice E** listam-se as classes das bibliotecas que constituem o Cliente-AS.

No **Apêndice F e Apêndice G** listam-se as classes das bibliotecas que constituem a duas versões da ABA CELIA. Respectivamente versão de comparação de semânticas de comunicação e versão de comparação de mobilidade.

2 Conceitos Básicos

Com esta secção pretende-se dar algumas explicações de suporte ao trabalho. Começa-se por fazer uma apreciação sobre o sistema Java, e define-se o que são agentes, sistemas que os suportam, e as aplicações baseadas em agentes.

2.1 Java

O Java [Sun98a] não é apenas mais uma linguagem de programação. É um sistema composto por uma linguagem, um conjunto vasto de bibliotecas de classes e ainda, um conjunto de ferramentas de suporte ao desenvolvimento de aplicações portáveis. Grande parte do poder do Java advém das suas bibliotecas de classes, chamadas de "packages", mas não só.

O potencial revolucionário do Java não é devido a uma característica principal, mas sim, devido a um conjunto de várias características como portabilidade, segurança e possuir uma linguagem orientada a objectos.

2.1.1 Portabilidade

A portabilidade de uma aplicação informática é a característica de se poder transferi-la de uma plataforma computacional para outra sem grandes dificuldades. Ou seja, a mesma aplicação pode ser executada independentemente da plataforma computacional de suporte, sem haver a necessidade de se proceder a alterações significativas no seu código fonte.

No Java a portabilidade é conseguida através de um conjunto de mecanismos. Um desses mecanismos é a transformação do código fonte em "bytecodes" que contêm instruções para um processador virtual (JVM). Depois esses "bytecodes" são interpretados por cada JVM e transformados em instruções específicas à máquina.

Outros mecanismos que contribuem para a portabilidade dos programas Java é o facto dos tipos de dados primitivos (caracter, inteiro, real, etc.) serem de tamanho fixo e a representação de código ser independente da rede (no formato *big-endian*).

A Figura 1 mostra o enquadramento da plataforma Java. Os programas Java são compilados para "bytecodes" e interpretados pela Java Virtual Machine que os transforma em instruções específicas à plataforma de Hardware.

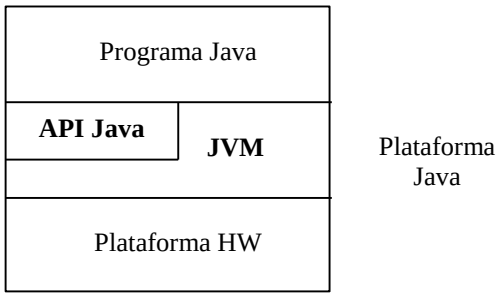

Figura 1: Plataforma Java

2.1.2 Segurança

A segurança é um dos aspectos primordiais em redes de computadores. O Java, concebido para ser utilizado em redes de computadores, também não descurou essa preocupação.

A segurança do sistema Java começa pela linguagem, que sendo robusta não permite a construção de programas incorrectos. Para isso contribui o facto de não possuir manipulação directa de memória (ponteiros), desta ser gerida automaticamente, e de controlar as conversões de tipos (casts) garantindo que não sejam violadas as regras da linguagem. Além disso são efectuados, pela máquina virtual, testes aos "bytecodes" gerados, quer na fase de carregamento do código, quer na fase de execução (verificação de limites de arrays, compatibilidade de tipos, etc.). Existe ainda um mecanismo de controlo de acessos a recursos (sistema de ficheiros local, ligações de rede, etc.). Esse controlo é feito por gestores de segurança. O nível de segurança pode ser ajustado consoante a origem dos "bytecodes".

2.1.3 Linguagem Orientada a Objectos

O paradigma de programação orientada a objectos corresponde a uma forma mais natural de resolver os problemas e possibilita a reutilização de componentes de aplicações. O foco desse paradigma são os dados (objectos) e a interface para sua manipulação.

O Java é uma linguagem orientada a objectos pura beneficiando das características daí inerentes. No entanto, não permite herança múltipla por razões de simplicidade e de desempenho.

Existem basicamente dois grupos de tipos: classes e interfaces. As classes correspondem a tipos abstractos ou concretos, com dados e métodos. As interfaces correspondem a tipos abstractos com a especificação de métodos, mas sem a sua implementação. A implementação dos métodos das interfaces fica a cargo das classes que implementam as respectivas interfaces.

2.1.4 Applets

Um *applet* é um mini programa que corre no contexto de um outro programa de maior dimensão. No caso do *applet* Java trata-se de uma mini aplicação que corre dentro de um Browser e que é carregado a partir de uma página *html*. Um *applet* Java é, assim, um programa em Java que pode ser incorporado numa página *html* praticamente da mesma forma que se incorpora uma imagem. Quando se utiliza um Browser compatível com Java para se visualizar a informação da página que contém o *applet*, o código do *applet* é transferido para a máquina local e executado pelo Browser.

2.1.4.1 Limitações

Por razões de segurança os *applets* carregados a partir da rede apresentam várias restrições. Uma delas é o facto de não poderem ler nem escrever na máquina local - o computador onde se estão a executar. Uma outra limitação é o facto de não poderem fazer ligações pela rede exceptuando as ligações para o próprio servidor donde foram carregados.

Esta limitação é facilmente superada devido à utilização do Voyager como infra-estrutura de comunicação. Para tal basta que exista um programa Voyager inicializado como servidor de *applets* na máquina do servidor Web, e que o *applet* tenha inicializado o Voyager em modo *applet*.

2.1.5 Outros Aspectos

Muito podia ainda ser dito acerca do Java, mas vamos referir aqui apenas alguns aspectos que são uma mais valia para o sistema Java.

- **Rapidez de desenvolvimento** – a facilidade de programação e semelhança com o C/C++ [Str91] do Java ajudam a produzir e "debbugar" código rapidamente. Especialmente aos programadores familiarizados com essas linguagens.

- **Garbage Collection** – permite eliminar alguns erros comuns de programação. Para destruir um objecto criado com new, basta simplesmente deixar de o referenciar. O Java encarrega-se de o destruir se verificar que não existem mais referências para ele.

- **Exception Handling:** facilidade de lidar com condições de erro. Um método que detecta uma condição de erro não usual atira uma excepção que pode ser apanhada e tratada de modo conveniente. Assim a execução pode prosseguir normalmente.

- **Multithreading:** Algumas aplicações podem executar múltiplas tarefas de forma cooperativa para uma maior eficiência de recursos.

- **Blibliotecas (packages):** mecanismo de agrupamento hierárquico de colecções de classes de Java e muito útil para a reutilização de código. Para além da organização e facilidade de gestão que proporciona, este mecanismo permite ainda

resolver o problema do conflito de nomes em código reutilizável. Por exemplo, as palavras "set", "get", "put", e "add" podem ser utilizadas por diversos programadores dando a origem a conflitos de nomes se eles reutilizarem código uns dos outros. A solução para isso é a existência do chamado "package prefix" no inicio do nome de cada classe. O nome `java.util.Hashtable.put()` define univocamente o método `put()` em questão.

- **Interfaces:** definição de métodos que um objecto suporta sem necessariamente fornecer a implementação desses métodos. Desde que o comportamento respeite o critério especificado, os detalhes da implementação dos métodos são irrelevantes.

2.2 Agentes de Software

Não existe consenso quanto à definição de agente de software. Todavia, agentes são componentes de aplicações que actuam em nome dos seus donos/utilizadores, apresentando um conjunto básico de atributos reconhecidos, tais como: autonomia, persistência e sociabilidade. Adicionalmente podem apresentar características como: inteligência, aprendizagem, ou mobilidade [S98].

Os agentes servem para facilitar "a vida" dos seus utilizadores, embora também tragam novos problemas e desconfianças [S98]. Através do modelo de delegação, os utilizadores atribuem aos seus agentes tarefas tipicamente rotineiras, complexas, demoradas, ou tarefas dificilmente realizáveis em tempo útil, por seres humanos.

Actualmente os agentes de software são o resultado da influência multidisciplinar de diferentes comunidades científicas, entre outras, das comunidades de Inteligência Artificial, Engenharia de Software e Interacção Homem-Máquina.

2.2.1 Agentes Móveis

Até hoje o principal paradigma de computação distribuída era baseado em objectos estacionários que comunicavam e interactuavam através da passagem de mensagens, principalmente síncronas. Este paradigma é incompleto e necessita de ser melhorado com mecanismos como mensagens assíncronas, mobilidade de objectos e objectos activos.

Os agentes móveis podem providenciar um paradigma de computação distribuída muito mais eficiente envolvendo sincronismo, assincronismo, passagem de mensagens, passagem de objectos, objectos estacionários e objectos móveis. Os objectos móveis são, então, partículas elementares para a computação de objectos distribuída.

Além da mobilidade, os agentes apresentam mais uma série de características computacionais básicas e importantes, tais como:

- **Passagem de objectos:** Quando um agente se move, o objecto todo é passado. Isto é, o seu código, dados, estado de execução, e seu itinerário.

- **Autonomia:** O agente contém informação suficiente para decidir para onde ir, fazer o quê, e quando ir.

- **Assincronismo:** O agente possui a sua própria "thread" de execução e pode ser executado assincronamente.

- **Interacção Local:** O agente móvel interage com outros agentes móveis ou com objectos estacionários localmente.

- **Operações desconectadas:** O agente móvel pode realizar as suas tarefas independentemente da rede estar ou não ligada.

- **Execução paralela:** Vários agentes podem ser enviados para diferentes locais para efectuarem tarefas em paralelo.

As vantagens dos agentes móveis são muitas, e não existe uma única alternativa isolada a toda a funcionalidade por eles proporcionada. Um dos interesses dos sistemas de agentes ou das aplicações baseadas em agentes prende-se com a possibilidade dos utilizadores destinarem tarefas aos agentes possibilitando computações assíncronas na Internet. Os agentes móveis são então as tais (ver Agentes de Software) entidades que representam os seus donos/utilizadores efectuando as tarefas mais ou menos complicadas e rotineiras que lhes são destinadas, percorrendo um maior ou menor número de máquinas até concretizarem a sua missão. Adicionalmente aos serviços de rede existentes, os agentes móveis possibilitam novos serviços e novos negócios.

2.2.2 Definição Adoptada no Âmbito do Trabalho

Muitas são as definições de agentes propostas, já que o conceito de agente é um paradigma e representa variadas áreas de aplicação com características próprias. Algumas dessas definições podem ser consultadas em [S98].

A noção de agente adoptada no âmbito deste trabalho corresponde à de [S98] e é definida segundo diferentes perspectivas, nomeadamente, segundo:

- **Perspectiva genérica**: Um agente é uma entidade software com uma identidade, estado e comportamento bem definidos, e representa, de alguma forma, o seu utilizador nas tarefas que realiza. Um agente deverá apresentar pelo menos as características de autonomia, sociabilidade, persistência, pró-actividade e reactividade.

- **Perspectiva técnica**: Um agente é um objecto activo de média granularidade. Isto significa que é uma instância de uma determinada classe, com o seu próprio grupo de actividades (*threads*), código e dados, e representado por um identificador único e global. Visto como um novo paradigma, o agente providencia o conceito de objecto activo e autónomo que pode ser adequado à concepção de aplicações dinâmicas, distribuídas e/ou complexas.

- **Perspectiva de utilização**: Para o utilizador, um agente pode ser visto como um novo paradigma de interacção homem-máquina, baseado no modelo de delegação, ou de gestão indirecta, por oposição ao modelo tradicional de interacção directa. O modelo de delegação é especialmente adequado às classes de aplicações emergentes na Internet, em actividades várias, caracterizadas por serem complexas, tediosas e/ou rotineiras, tais como: a pesquisa de informação em espaços vastos e pouco estruturados; gestão de correio electrónico; ou comércio electrónico.

Apesar da inteligência e mobilidade não serem fundamentais para se definir uma entidade de software como agente, atendendo às características da infra-estrutura AgentSpace, a mobilidade é uma característica suportada de forma a permitir, entre outras funcionalidades: computação desconectada; tolerância a falhas; e interacções distribuídas mais eficientes. Quanto à inteligência, não constitui claramente o foco e interesse do trabalho.

2.3 Sistemas de Suporte de Agentes

Um SSA é uma sistema de gestão e de suporte à execução de agentes. Ele é responsável por providenciar, para além de um ambiente computacional de execução/interpretação de agentes, também mecanismos de navegação, comunicação, segurança e persistência entre outros. O SSA tem ainda de providenciar interfaces (API) específicas de acesso a serviços e recursos externos, tais como bases de dados, sistemas de ficheiros ou dispositivos físicos.

Designa-se por **contexto** a interface providenciada por cada SSA. Um contexto é representado por um endereço electrónico que identifica univocamente a existência de um determinado SSA. Um contexto pode encontrar-se organizado através de um conjunto lógico de entidades, designadas por **locais de execução**, ou simplesmente locais. Os locais de execução são pontos lógicos onde os agentes se executam, encontram e comunicam com outros agentes. Cada local é identificado univocamente por um endereço electrónico.

2.4 Aplicações Baseadas em Agentes

As Aplicações Baseadas em Agentes são aplicações que de um modo geral pressupõem a existência de vários agentes que interactuam e comunicam entre si, embora nada impeça que sejam constituídas apenas por um agente.

A metáfora adequada às ABA é a comunidade de agentes. As comunidades de agentes podem ser homogéneas ou heterogéneas. Uma comunidade homogénea é formada por um conjunto de agentes que partilham o seu conhecimento e que sabem como comunicar numa linguagem comum. Consiste no conjunto contextualizado de agentes, baseados sobre um Sistema de Suporte de Agentes (SSA) comum. O conceito de comunidade heterogénea é uma evolução, em termos de complexidade e de capacidades desejáveis, relativamente ao conceito de comunidade homogénea.

Consiste no conjunto contextualizado de agentes, mas potencialmente baseados sobre distintos SSA, ou seja, pressupõe a comunicação entre agentes heterogéneos.

2.5 ORB – Object Request Broker

Um Object Request Broker (ORB) é um sistema que providencia a interoperação entre objectos distribuídos. Um ORB providencia, entre outros, serviços básicos como: criação de objectos que podem ser invocados remotamente, obtenção de referências para objectos remotos, envio de mensagens e invocação de métodos em objectos remotos, gestão de nomes, e gestão de interfaces.

2.5.1 CORBA

A especificação CORBA (Common Object Request Broker Architecture) [OMG91] da OMG é uma especificação que privilegia a interoperação entre objectos desenvolvidos segundo diferentes linguagens de programação e existentes em diferentes arquitecturas e sistemas operativos. Como especificação que é, não existe qualquer sistema "CORBA". O que existem são diferentes produtos que implementam a especificação CORBA.

A arquitectura DCOM/OLE (Distributed Component Object Model / Object Linked and Embedding) [MD94] da Microsoft, privilegia a interoperação entre objectos desenvolvidos segundo diferentes linguagens de programação para os sistemas operativos Windows.

2.5.2 RMI

Para a interoperacionalidade de objectos o Java apresenta um serviço de objectos distribuídos designado por RMI (Remote Method Invocation) [Sun97b]. No entanto, esta proposta pretende fornecer uma infra-estrutura para interoperação de objectos distribuídos desenvolvidos em Java, contrariamente aos objectivos do CORBA e DCOM, em que se prevê a interoperacionalidade entre objectos de múltiplas linguagens de programação e sistemas operativos. Como a interoperacionalidade oferecida pelo RMI é entre objectos da mesma linguagem, os mecanismos providenciados ao programador são significativamente mais simples, elegantes e por conseguinte, mais fáceis de utilizar.

2.5.3 Voyager

O Voyager é um produto comercial da empresa ObjectSpace [Obj97a] para suporte ao desenvolvimento de aplicações Java distribuídas. O Voyager é um ORB "100% Java" integrado de forma adequada com o JDK 1.1. O Voyager pode ser visto como um ORB CORBA tradicional, mas com um conjunto de especificidades adicionais.

Como estrutura de comunicação o Voyager providencia funcionalidades para:

- Criação de objectos que podem ser invocados remotamente.

- Obtenção de referências para objectos remotos.

- Envio de mensagens e invocação dinâmica de métodos em objectos remotos. Evidentemente, essas operações também se aplicam a objectos locais (no Java normal).

Adicionalmente, o Voyager providencia outras funcionalidades incluindo:

- Persistência: os objectos podem sobreviver aos programas que os criam.

- Reciclagem automática e distribuída de objectos.

- Um mecanismo de comunicação flexível baseado em mensageiros inteligentes e com uma variedade de métodos de comunicação entre objectos (síncrona, assíncrona diferida, assíncrona num único sentido, e num único sentido com difusão selectiva).

- Serviço distribuído de nomes.

- Mobilidade de objectos: De modo que estes possam ser trocados entre diferentes programas Java.

- Suporte a eventos distribuídos.

- Mecanismo de publicação/subscrição de mensagens.

- Comunicação em grupo.

- Integração adequada com a Web: possibilitando a interoperação de applets com objectos remotos.

- Interoperação com objectos CORBA no servidor.

O Voyager pode ser comparado com outros ORB Java como por exemplo o RMI da Sun, sendo claramente vantajoso por ter mais funcionalidades, e ser mais rápido, elegante e simples de aprender e utilizar.

3 Técnicas

3.1 Processo de desenvolvimento

3.1.1 Prototipagem

A metodologia de desenvolvimento de Software que utilizamos no desenvolvimento do Cliente-AS bem como das Aplicações Baseadas em Agentes foi a Prototipagem.

A Prototipagem, segundo Roger S. Pressman [P92], é um processo que permite a quem desenvolve o projecto a criação rápida de um modelo do software a ser construído. Esse modelo pode apresentar-se numa de três formas:

1. Um protótipo em papel ou baseado em computador que exemplifica a interacção homem-máquina de tal forma que o utilizador possa compreender como é que esta ocorre.

2. Um protótipo funcional que implementa um subconjunto das funcionalidades pretendidas.

3. Um programa existente que faz parte de toda a funcionalidade desejada, mas que possui outras características que vão ser melhoradas no processo de desenvolvimento.

O protótipo criado, o qual deve ser melhorado, ou mesmo abandonado, é uma forma de aprofundar o conhecimento do problema a tratar, servindo assim de mecanismo de identificação de requisitos e do seu refinamento. Esse protótipo é então um esboço feito com uma certa rapidez, mas que vai sendo ciclicamente afinado, ao mesmo tempo que ajuda a quem desenvolve o projecto a compreender o que é que precisa ser feito.

Inicialmente desenvolvemos um protótipo que apenas implementava um subconjunto das funcionalidades. Nessa altura fazia sentido usar o *UI Designer* do JBuilder para desenvolver rapidamente esse protótipo. Posteriormente, verificou-se que a utilização do *UI Designer* limitava a reutilização de classes de janelas, sendo mais adequado para a construção de raiz dessas janelas. Esta desvantagem é decisiva se considerarmos que as janelas do Cliente-AS são instâncias de classes que estendem uma única classe abstracta que colecciona um conjunto de funcionalidades comuns a todas elas.

Uma outra alteração ao protótipo inicial, foi a utilização de mais classes da biblioteca `java.awt`, em substituição das classes das bibliotecas do JBuilder, por uma questão de diminuição do tamanho do ficheiro que é carregado juntamente com o *applet* melhorando assim a rapidez do seu carregamento.

3.1.2 Reutilização

Em qualquer uma das fases do projecto de desenvolvimento de software a reutilização deve ser sempre um objectivo a atingir.

No que respeita ao desenvolvimento do Cliente-AS a reutilização foi usada intensivamente no desenho e implementação. Essa reutilização é feita recorrendo a componentes abstractas e utilizando mecanismos de herança (quer utilizando componentes definidas nas bibliotecas de classes do Java e JBuilder, quer utilizando componentes definidas nas bibliotecas de classes do Cliente-AS, sendo estas geralmente classes abstractas que têm de ser estendidas).

Exemplo de classes abstractas definidas no cliente são: `ManageList`, `Column`, `ClassDetails`, `ASClientDialog`, entre outras. Além da utilização de classes abstractas como forma de reutilização, houve ainda a preocupação de potenciar a reutilização de classes que definem objectos específicos. Por exemplo, a classe que modela a lista de agentes disponíveis é definida a partir da classe da lista de agentes de um determinado utilizador, acrescentando apenas informação sobre o dono do agente. Estas classes são, respectivamente, `ShowAvailableAgentsList` e `ManageMyAgentsList`.

3.2 Ferramentas utilizadas

3.2.1 JDK1.1

O JDK (Java Development Kit) [Sun98b] é uma ambiente de desenvolvimento de *applets* e de aplicações Java. Esse ambiente é constituído por um compilador, um interpretador Java, bibliotecas standard de classes, e um conjunto de outras ferramentas auxiliares que invocadas a partir da shell do sistema operativo, já que não é fornecida uma interface gráfica para o utilizador. Das suas ferramentas destacam-se o gerador de documentação "javadoc", o gerador de compilações de classes "jar" e o "appletviewer" útil para teste de *applets*, entre outras.

3.2.2 JBuilder

O JBuilder é um produto da Inprise, destinado ao desenvolvimento de aplicações Java. Trata-se de um ambiente de desenvolvimento com várias ferramentas tais como: *Wizards* (úteis para criação de applets, projectos, geração de compilações de classes, etc), interfaces gráficas de utilizador (*UI Designer*). Outra funcionalidade muito útil proporcionada por este ambiente, é o mantimento um ficheiro de dependências para a compilação de projectos de grande dimensão.
Foi utilizada a versão 1.01 deste produto.

Este ambiente de desenvolvimento foi utilizado na implementação do Cliente-AS e das ABA desenvolvidas nomeadamente a aplicação CELIA.

3.2.3 Voyager

O Voyager pode ser encarado como sendo um ambiente de desenvolvimento. Uma ferramenta utilizada no desenvolvimento do trabalho foi o vcc.

O vcc (virtual class creator) [Obj97a] é uma ferramenta do Voyager. O objectivo dessa ferramenta é a criação de classes que possam ser referenciadas remotamente. Para isso a ferramenta produz uma versão virtual para as classes que se pretendem que sejam acedidas remotamente. A ferramenta recebe como entrada a classe(s) local(ais) e produz a(s) respectiva(s) versão(ões) virtual(ais). Cada classe virtual fica com o mesmo nome, igual ao nome original excepto a primeira letra que é um "V". A partir dessas classes virtuais é então possível criar instâncias remotas da(s) classe(s) respectivas. Desta forma o código da classe não necessita de ser alterado.

O vcc pode ser utilizado de igual forma em interfaces. Neste caso a versão virtual da interface implementa a interface original e estende `VObject`. Pode ainda ser utilizado em classes abstractas e em classes não públicas. No entanto, nestes casos as versões virtuais não podem ser utilizadas para construir instâncias.

Em relação ao nosso trabalho, a ferramenta vcc foi utilizada na criação de uma classe virtual usada no mecanismo de notificação de eventos.

3.3 A Infra-estrutura AgentSpace

O AgentSpace [SMD98], ou Espaço de Agentes em português, é um Sistema de Suporte de Agentes (SSA), ou seja, um sistema de gestão e suporte à execução de agentes. Desta forma, como se verifica na Figura 2, o AgentSpace consiste numa infra-estrutura para suporte, desenvolvimento e gestão de ABA.

Como SSA que é, o AgentSpace tem como objectivos principais o suporte, o desenvolvimento e a gestão de aplicações dinâmicas e distribuídas baseadas em agentes. Estes objectivos são concretizados através de três componentes integradas:

1. **Servidor AgentSpace (Servidor-AS)**: Consiste em um processo Java, de múltiplas actividades, no qual os agentes podem ser executados eficientemente e em segurança. O Servidor-AS providencia vários serviços, designadamente e entre outros: criação de agentes e locais de execução, execução de agentes, persistência, controlo de acessos, suporte à mobilidade e comunicação de agentes, geração de identificadores, e opcionalmente uma interface simples (*shell*) de gestão e monitorização.

2. **API AgentSpace (API-AS)**: Consiste em uma biblioteca de classes e interfaces, que definem as regras de como se desenvolvem (classes de) agentes, em particular, e aplicações baseadas em agentes, em geral. Nomeadamente, a API-AS suporta o programador na construção de classes de agentes que são criadas e armazenadas no Servidor-AS para posterior utilização.

3. **Cliente AgentSpace (Cliente-AS)**: Foco deste trabalho, consiste num *applet* Java que permite a gestão e monitorização de agentes e de outros recursos existentes, de forma integrada com a Web/Internet. Oferece a qualquer utilizador, salvaguardando questões de controlo de acessos e de segurança, a possibilidade de gerir facilmente os seus próprios recursos mantidos em um ou mais Servidores-AS.

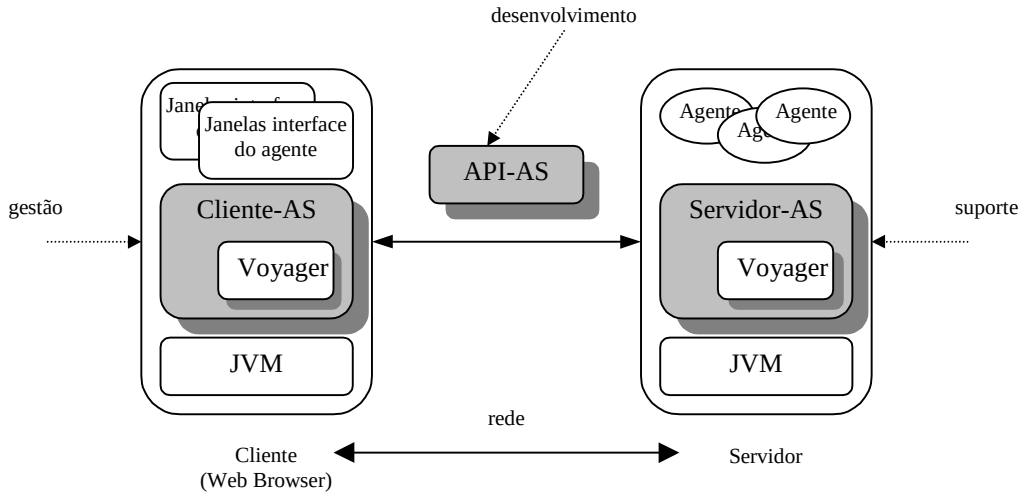

Figura 2: Visão geral da infra-estrutura AgentSpace.

3.3.1 API-AS

A API do AgentSpace é um pacote de interfaces e classes em Java que ajudam o programador na construção de classes de agentes e de *applets* cliente. Estes clientes podem ser aplicações não específicas, como o Cliente-AS, ou aplicações específicas como alguma aplicação de comércio electrónico.

Seguidamente apresentam-se alguns conceitos básicos da API do AgentSpace que definem as regras de construção de agentes ou de aplicações baseadas neles:

- **`Agent`** – Um agente pode ser executado em vários locais de execução, locais ou remotos. A identificação do agente é dada por `AgentId`. O Agente tem informação que identifica o seu dono. Para a criação de uma classe agente, o programador deve estender a classe abstracta `Agent`. A classe `Agent` possui quatro grupos de métodos:

  - Finais – operações predefinidas para todas as classes de agentes. Exemplos: `moveTo()`, `save()`, `die()`, `clone()`, `getId()`.

  - Callbacks – métodos que podem ser redefinidos por classes de agentes específicas e que são invocados transparentemente pelo Servidor-AS como

resultado de um evento. Exemplos: `run()`, `onCreation()`, `beforeMove()`,`handleMessage()`.

- Helper – métodos com modificador de acesso *protected* ou *private* usados para suportar certas funções.

- **`Place`** – define um local de execução, metáfora conceptual de programação para designar os sítios onde os agentes se executam e se relacionam entre si. Define os níveis de controlo de acesso, e permite o controlo dos recursos computacionais. Os locais de execução contêm o número máximo permitido de agentes, bem como o seu número corrente, de forma a suportar uma gestão de recursos elaborada. A identificação do local é dada por `PlaceId`.

- **`AgentView`** – uma interface que providencia o acesso aos agentes. O acesso é feito indirectamente via `AgentView` de forma a protegê-los, e para esconder transparentemente as suas localizações correntes. Além disso o `AgentView` evita a necessidade de criar e gerir classes remotas/virtuais. Esta interface em Java é gerida pelo núcleo do Servidor-AS que possui informação relativa a agentes, nomeadamente: o agente envolvido, a sua identidade (aid), o seu local de execução corrente e nativo, a identificação do seu dono, e a identificação da sua classe de agente.

- **`PlaceView`** – uma interface que providencia o acesso controlado ao local de execução.

- **`ContextView`** – uma interface que fornece uma referência para o contexto do Servidor-AS, que é o seu objecto mais importante e crítico. O objecto `Context` contém uma lista de todos os outros objectos envolvidos, nomeadamente: places, `users`, `groups`, e lista de controlo de acesso. O `ContextView` providência o acesso controlado ao contexto do Servidor-AS. Consoante a informação de autenticação do utilizador, o objecto `ContextView` activa ou desactiva um conjunto de operações gerais. Exemplos dessas operações são: `createPlace()`, `remotePlace()`, `getPlaceOf()`, `places()`, `createUser()`,`users()`,`groups()`, etc.

- **`User`** – um humano, uma organização, ou mesmo outro agente que pretende que um determinado agente lhe execute determinada(s) tarefa(s). Um utilizador é identificado pelo seu país, empresa e nome.

- **Controlo de acessos** – dado por uma estrutura de dados que mantém o nível de acesso de cada utilizador. Pode-se também criar grupos de utilizadores com o mesmo nível de acesso. Há quatro graus de acesso correspondentes a tipos de utilizador:

  - Anónimo – grau de acesso mais baixo.

  - Donos de agentes – corresponde a um grau de acesso que permite operações efectuadas sobre os seus agentes.

  - Donos de locais de execução – corresponde a um grau de acesso que permite operações efectuadas sobre os seus locais de execução.

  - Administradores – grau de acesso mais elevado.

- **`ASId`** – um objecto desse tipo identifica/endereça univocamente um Servidor-AS. Possui a seguinte informação:

  - DNS ou endereço IP da máquina

  - Porto de contacto do servidor
    Exemplo: "*cupido.inesc.pt:777*"

- **`PlaceId`** – o seu objecto identifica/endereça univocamente um local de execução. Possui a seguinte informação:

  - Identidade do `ASId`

  - Nome único baseado num contador sequencial
    Exemplo: "*cupido.inesc.pt:777/pid001*","*197.0.123.22:777/pid023*"

- **`AgentId`** – o seu objecto identifica/endereça univocamente um agente. Possui a seguinte informação:

  - Identidade do `PlaceId` onde o agente foi criado

  - Nome único baseado num contador sequencial
    Exemplo:
    "*cupido.inesc.pt:777/pid001|aid002*","*197.0.123.22:777/pid023|aid033*"

3.4 Desenho e Implementação do Cliente-AS

Uma forma simples de desenvolver o Cliente-AS é utilizar *html* para construir uma parte da interface com o utilizador, através do uso de *links* que corresponderiam às várias opções disponíveis, substituindo assim as janelas tradicionais, por páginas *html* com *applets* específicos destinados à manipulação de informação e comunicação com o Servidor-AS. Esta forma de implementação do cliente embora fosse menos pesada, não é muito eficaz, já que a adaptação do Cliente-AS ao perfil do utilizador é mais difícil, juntamente com o controle de acessos efectuado nas diversas opções correspondentes às operações do servidor.

Decidimos utilizar apenas um *applet*, cuja função destina-se apenas a criar uma janela com menus (instância de uma subclasse de `Frame`), através dos quais conseguimos providenciar um sistema de janelas muito mais fácil de gerir e utilizar.

As operações que o Cliente-AS deve permitir encontram-se divididas em quatro grupos correspondente ao perfil típico do utilizador que as executa:

- **Operações de utilizadores anónimos:** Neste grupo encontram-se as operações de interacção com os agentes disponíveis (agentes cuja política de segurança o permita).

- **Operações de donos de agentes:** As operações que se encontram neste grupo permitem gerir os agentes do utilizador, bem como consultar as suas classes e políticas de segurança.

- **Operações de donos de locais de execução:** Estas operações permitem a gestão dos locais de execução e suas políticas de segurança respectivas.

- **Operações de administradores:** Tal como gestão de utilizadores, grupos de utilizadores, introdução/remoção de novas classes de agente bem como de políticas de segurança. Permite também configurar os recursos do Servidor-AS.

Para mais detalhes sobre a especificação do Cliente-AS consultar [S98].

O Cliente-AS é um *applet* que é carregado a partir de uma página *html*. O único requisito necessário é a existência de um cliente Web (browser) que suporte Java, nomeadamente JDK1.1. A partir desse *applet* obtém-se uma janela definida por uma classe que estende `Frame` e que permite o acesso a todas as funcionalidades do cliente utilizando para isso janelas filhas cuja classe estende `Dialog`.

Na Figura 3 ilustra-se todo o processo. Juntamente com a leitura da página *html* que contém o *applet*, o cliente Web efectua o carregamento do arquivo *ASClient.jar*. Este arquivo é composto por:

- **Bibliotecas de classes do Cliente-AS**.

- **Bibliotecas de classes da infra-estrutura de comunicação Voyager**.

- **Bibliotecas de classes fornecidas pelo JBuilder:** Apenas as classes referenciadas pelo Cliente-AS e as que são referenciadas por estas.

- **Bibliotecas de classes da API-AS**.

Após o carregamento, o cliente Web inicia a execução do *applet*. Este cria e mostra a janela principal da aplicação e abre uma janela de identificação e estabelecimento de ligação. O utilizador deverá identificar-se fornecendo a sua identificação e palavra-chave que determinarão o seu perfil de utilizador.

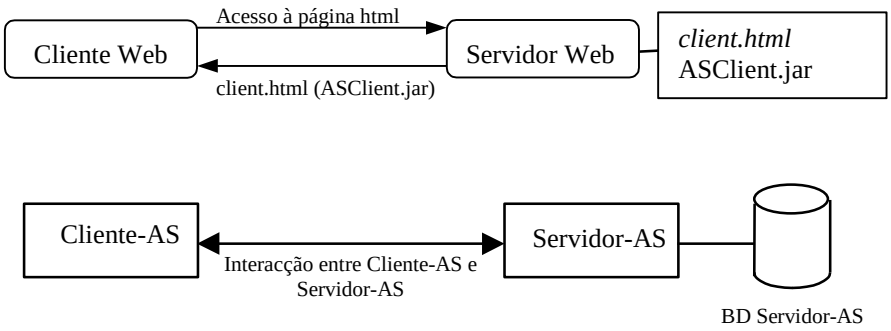

Figura 3: Inicialização do Cliente-AS

A Figura 4 mostra como é feita a interacção entre o Cliente-AS e o Servidor-AS. Como já foi dito o *applet* ao ser carregado vem também com as classes que compõem a API do AgentSpace, e as classes do Voyager. As classes API-AS são necessárias porque fornecem a interface entre cliente e servidor, enquanto que as classes do Voyager servem como mecanismo de comunicação (e de ORB). Inerente à utilização da linguagem Java está a necessidade de existir uma versão de Java Virtual Machine numa camada inferior, neste caso JDK1.1.

Figura 4: Interacção entre Cliente-AS e Servidor-AS

3.4.1 Bibliotecas do Cliente-AS

As classes do Cliente-AS estão agrupadas em várias bibliotecas, consoante a sua função. Assim as classes relativas ao `applet` ficam numa biblioteca diferente das classes relativas à ligação com o Servidor-AS, ou mesmo das classes que definem a janela principal do Cliente-AS.

De seguida apresenta-se a descrição das bibliotecas:

- **inesc.as.client:** Classes genéricas usadas na implementação de diversas funcionalidades do cliente, tal como certas excepções, constantes, e classes que possibilitam a serialização de enumerados.

- **inesc.as.client.applet:** Classes relativas ao *applet*.

- **inesc.as.client.connection:** Contém uma classe que encapsula dados relativos a uma ligação, tal como a referência à interface do servidor (`ContextView`) e uma ao utilizador corrente. Contém ainda a informação inserida pelo utilizador quando este liga-se a um servidor, tal como o a sua identificação, palavra-chave, identificação do servidor, etc. Contém também uma classe excepção de ligação, uma classe de evento de ligação e uma interface que permite às classes que a implementam receber eventos sobre ligações.

- **inesc.as.client.component:** Contém classes que definem componentes usadas pelas janelas do Cliente-AS

- **inesc.as.client.panel:** Contém classes dos paineis usadas pelas janelas do Cliente-AS.

- **inesc.as.client.window:** Contém classes relativas às janelas do Cliente-AS tal como uma classe que calcula a posição de uma janela, uma interface que indica se uma janela é passível de impressão, um gestor de janelas, uma interface usada para saber quando é que uma janela fica activa, etc.

- **inesc.as.client.error:** Classes relacionadas com o tratamento de erros. Contém uma interface que permite receber a notificação que um erro aconteceu.

- **inesc.as.client.window.main:** Classes relacionadas com a janela principal da aplicação, tal como a classe da janela propriamente dita, a classe da barra de menu e a classe da barra de estado.

- **inesc.as.client.window.dialog:** Contém as classes que definem as janelas filhas do Cliente-AS juntamente com classes auxiliares como, por exemplo, uma interface usada receber eventos que indicam o motivo de uma janela ter sido fechada.

- **inesc.as.client.multicolumnlist:** Contém a classe que define uma lista genérica com várias colunas, bem como as classes que a estendem.

- **inesc.as.client.multicolumnlist.column:** Contém a classe que define uma coluna genérica usada pela lista referida anteriormente. contém também as colunas específicas.

3.4.2 Gestor de ligações

De forma a encapsular os dados relativos a uma ligação entre o Cliente-AS e um Servidor-AS, bem como gerir essa conexão foi desenvolvida a classe `Connection`. Esta classe permite iniciar, ou terminar uma ligação, bem como alterar o utilizador corrente, ou alterar a palavra-chave deste. Uma ligação pode dar origem a excepções se, por exemplo, efectuarmos a operação que altera a palavra-chave e o Cliente-AS não estiver ligado a nenhum Servidor-AS. Essas excepções são definidas pela classe `ConnectionException`.

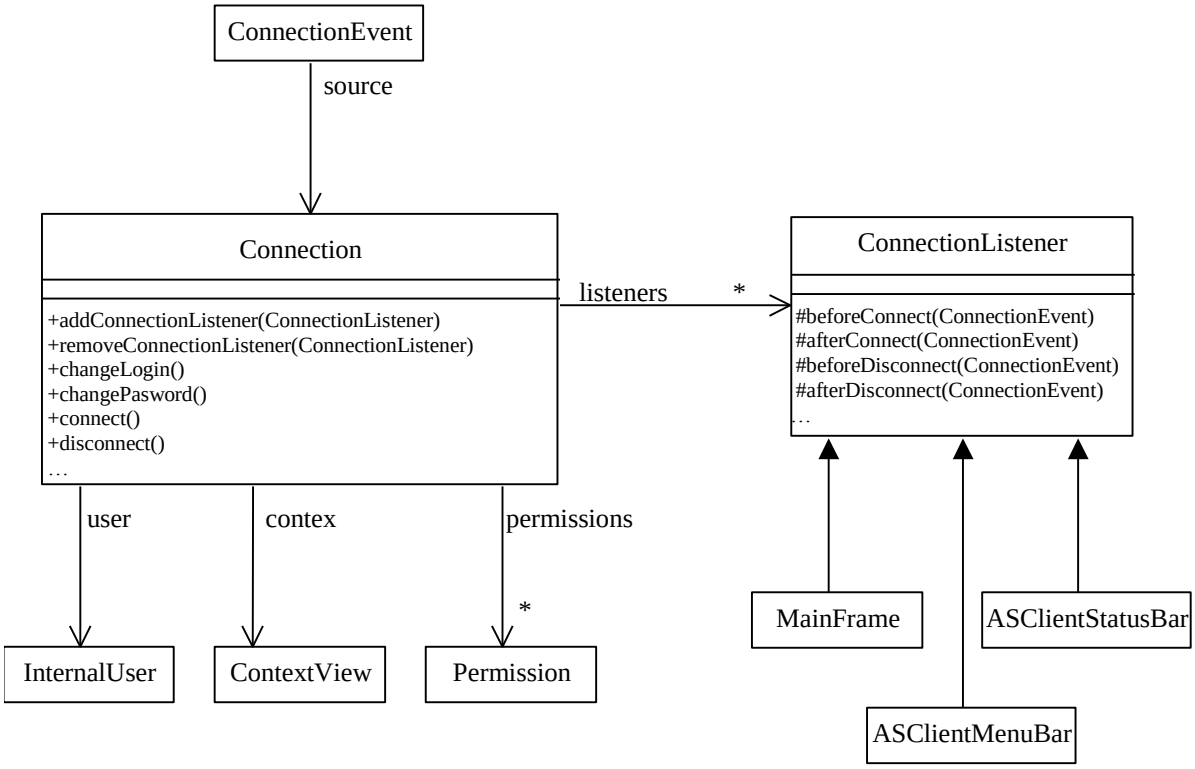

Figura 5: Gestor de ligações - diagrama de classes

Uma ligação pode também dar origem a eventos, como o inicio e fim de uma ligação, os quais estão representados pela classe `ConnectionEvent`. Uma classe que implemente a interface `ConnectionListener` pode registar-se de modo a receber esses eventos (vid. Figura 5 para o diagrama de classes do gestor de ligações). Exemplos dessas classes são a barra de estado, bem como o menu da janela principal do Cliente-AS.

3.4.3 Janelas do Cliente-AS

3.4.3.1 Janela principal

A janela principal desta aplicação, ilustrada na Figura 6, é construída pelo *applet* e é definida pelas três classes que se apresentam de seguida e que se encontram na biblioteca `inesc.as.client.window.main`:

- **`MainFrame:`** define a janela propriamente dita. Estende a classe `DecoratedFrame` e implementa a interface `ConnectionListener`, de forma a saber, por exemplo, se uma ligação foi terminada, fechando assim todas as janelas relativas a essa ligação. Implementa também `ErrorListener`, de forma que possa notificar o utilizador da ocorrência de erros.

- **`ASClientStatusBar:`** Estende `StatusBar` e define a barra de estado da aplicação. Tal como a classe `MainFrame` implementa `ConnectionListener` e `ErrorListener`. Além disso implementa `ASClientDialogListener`, permitindo assim notificar o utilizador das razões pela qual uma janela foi fechada – embora na versão corrente do Cliente-AS essa informação não seja usada, devido a não ser muito importante para o utilizador.

- **ASClientMenuBar:** Estende `MenuBar` e define a barra de menus da aplicação. Implementa `ConnectionListener`, permitindo assim a activação ou desactivação das opções do menu, consoante exista ou não uma ligação activa. Como certas opções do menu, tais como "print" e "close" dependem de haver ou não uma janela filha activa, esta classe implementa `ActiveWindowListener`.

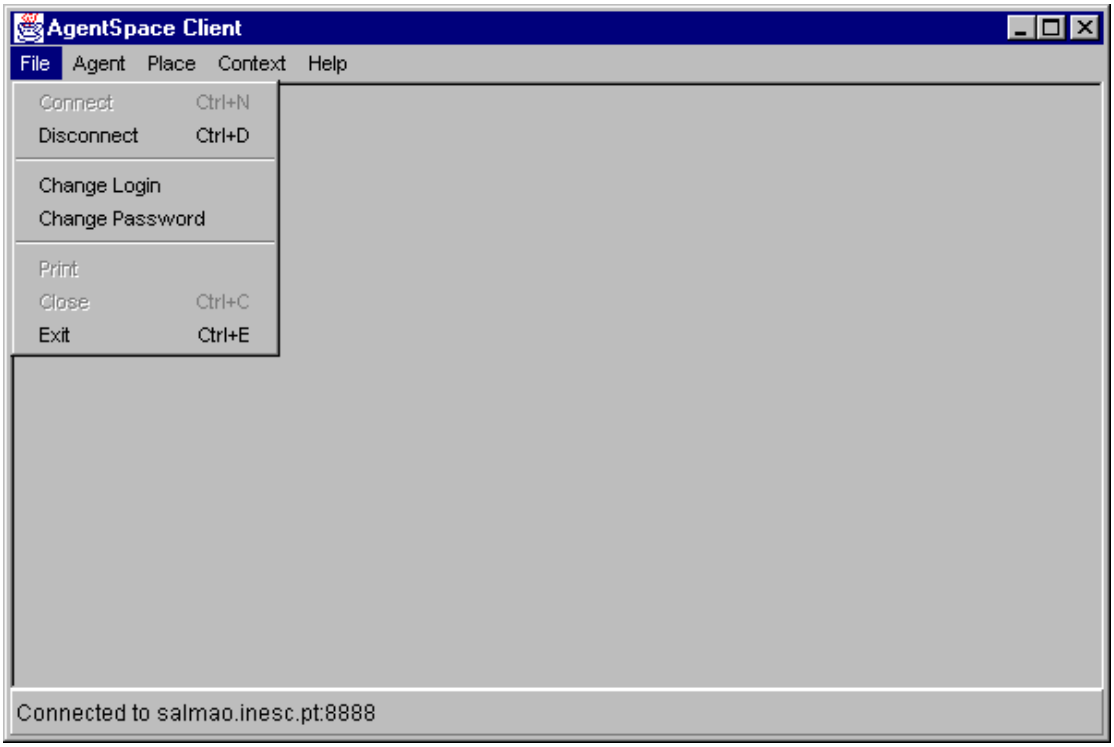

Figura 6: Janela principal do Cliente-AS

As janelas filhas desta aplicação são todas instâncias de classes que estendem a classe abstracta `ASClientDialog`. Estas janelas originam eventos, que são recebidos por objectos que implementam a interface `ASClientDialogListener`. Esses eventos são originados quando uma janela é fechada, indicando o motivo. Deste modo um objecto que implemente essa interface e tenha-se registado sabe se uma janela foi fechada porque o utilizador fechou a janela, ou porque este premiu o botão "Ok" aceitando assim as operações efectuadas ou ainda se "Cancel" foi premido, ignorando assim as alterações efectuadas nessa janela. Esta classe define ainda as dimensões máximas da janela. O diagrama de classes é apresentado na Figura 7.

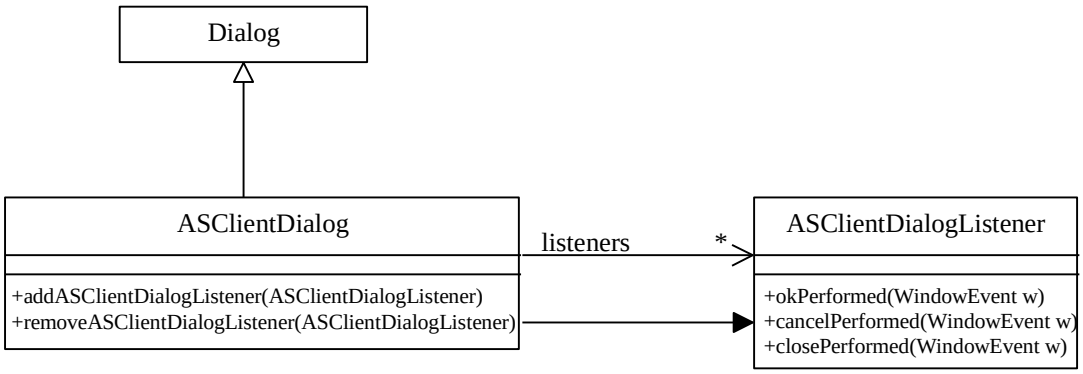

Figura 7: Classe genérica das janelas filhas do Cliente-AS - diagrama de classes

A localização de uma janela, é definido por uma instância da classe `WindowLocation` que estende `Point` (vid. Figura 8). Essa classe permite a especificação do alinhamento pretendido em relação a outra componente. Por exemplo, a janela de visualização/configuração das propriedades de um agente aparece no canto superior esquerdo da janela principal, enquanto a janela que permite mudar o utilizador corrente aparece no centro.

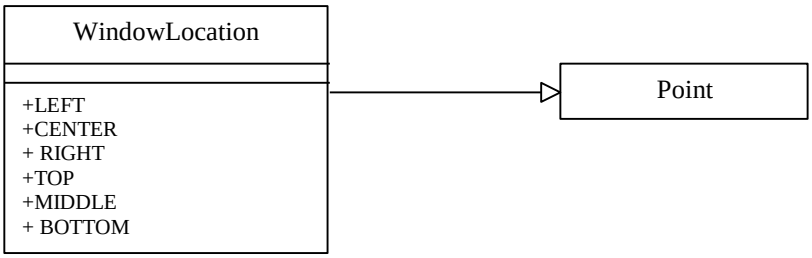

Figura 8: Classe que define a posição de uma janela

As janelas que podem ser imprimidas implementam a interface `Printable`. Deste modo a opção "print" do menu está ou não activa consoante a janela implemente ou não essa interface.

3.4.3.2 Janelas de criação e configuração

As janelas de configuração e criação de objectos partilham certas características, tais como a possibilidade de aceitar ou cancelar as operações efectuadas, de, para efectuar essas operações buscarem informação que se encontra distribuída pelas várias componentes da janela, e para além disso, normalmente a informação se relaciona com um mesmo assunto encontrar-se agrupada.

Devido a estas características, as janelas dessa natureza são instâncias de classes que estendem a classe abstracta `CreateDialog`. Esta classe permite a introdução de grupos de informação, através do método `addGroupBox()`. As componentes que

guardam a informação introduzida devem implementar a interface `Parameter()` e definir o método `getParameter()` que permite o acesso ao valor introduzido independentemente da classe da componente. Exemplo de classes que implementam essa interface são:

- **`ASClientChoice:`** Estende `Choice`, e devolve o item seleccionado no método `getParameter()`.
- **`ASClienJCSpinBox:`** Estende `JCSpinBox`.
- **ASClientTextField:** Estende TextField.

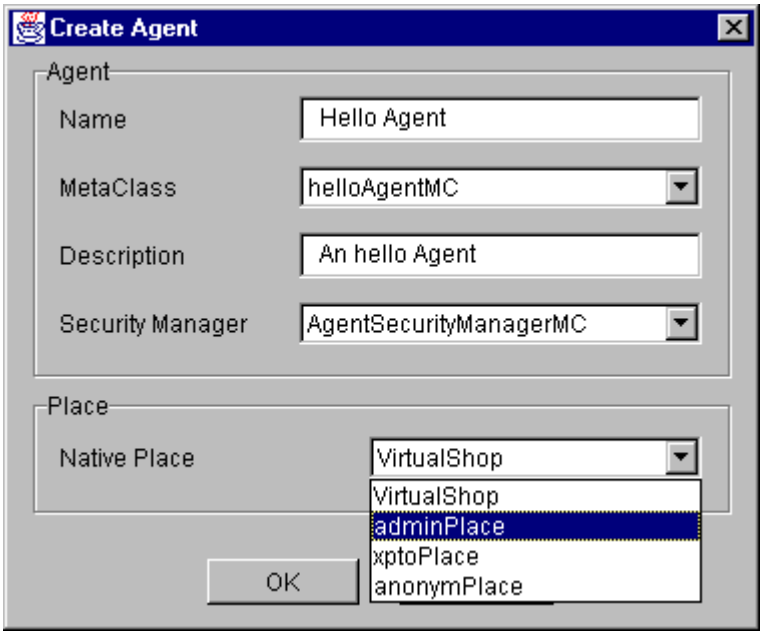

Figura 9: Janela de criação de agentes

As classes que estendem `CreateDialog`, como por exemplo a classe que define a janela de criação de um agente (vid. Figura 9), apenas têm que definir o método abstracto `create()` invocado quando o utilizador pressiona o botão "ok". Os parâmetros necessários são obtidos automaticamente, através de `getParameter()` especificando o nome do grupo de informação e o nome do campo ao qual se quer ir buscar o valor do parâmetro. Esta classe distingue os campos necessários dos opcionais, permitindo assim a activação ou desactivação do botão "ok".

3.4.3.3 Janelas informativas

As classes que definem as janelas que fornecem informação sobre algum objecto estendem `DetailDialog`. Esta classe permite organizar a informação em grupos e permite que a exposição de informação seja facilmente definida pelas classes que a estendem.

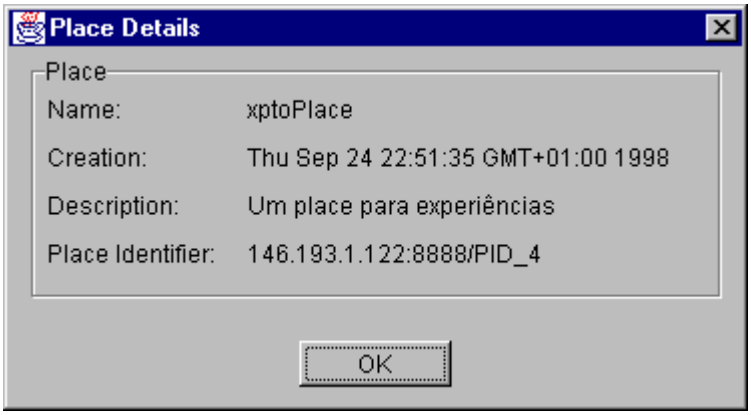

Figura 10: Janela de informação sobre um local de execução

A Figura 10 mostra uma destas janelas. A classe `PlaceDetailsDialog` que define esta janela estende `DetailDialog` e mostra informação detalhada sobre um local de execução. A data de criação do local actualmente encontra-se formatada para o local onde o *applet* está a ser executado, no entanto a imagem refere-se a uma versão anterior.

3.4.3.4 Janelas com uma lista de itens

Por vezes para efectuar operações sobre um objecto, é necessário seleccioná-lo de um conjunto de informação do mesmo tipo. Assim optou-se por apresentar essa informação na forma de uma lista, em que é possível seleccionar-se uma linha. Essa linha deve mostrar informação sobre o objecto que se quer manipular. Como essa informação é normalmente estruturada, usamos uma lista com várias colunas.

Normalmente uma impressão dessa janela será útil para o utilizador, por isso estas janelas implementam a interface `Printable` de modo a activar a opção "print" do menu quando uma destas janelas está activa.

As classes principais que definem essa janela são:

- **`ManageDialog`:** Classe que define a janela propriamente dita. Tem métodos que simplificam a criação de botões, utilizados pelas classes das janelas que a estendem. Simplifica ainda a activação/desactivação de botões, consoante exista ou não um item seleccionado na lista definida pela classe `ManageList`.

- **`ManageList`:** Classe que define a lista com várias colunas. As classes que a estendem só precisam de adicionar as colunas que desejam, através do método `addColumn()` e definir o método `getKeys()` que devolve as chaves utilizadas para buscar a informação que preenche as colunas. Esta classe implementa `Enumeration`, sendo as sucessivas linhas devolvidas pelo método `nextElement()`.

- **`Column:`** Representa uma coluna que pode ser adicionada a uma lista que estenda `ManageList`. As classes que estendem `Column` só precisam de definir o método `getTitle()` que devolve o título da coluna, e o método `fetchCell()` que devolve o conteúdo de uma coluna numa determinada linha a partir da chave que recebe como parâmetro. Esta classe implementa `Enumeration`, devolvendo o método `nextElement()`, o conteúdo de uma coluna linha a linha.

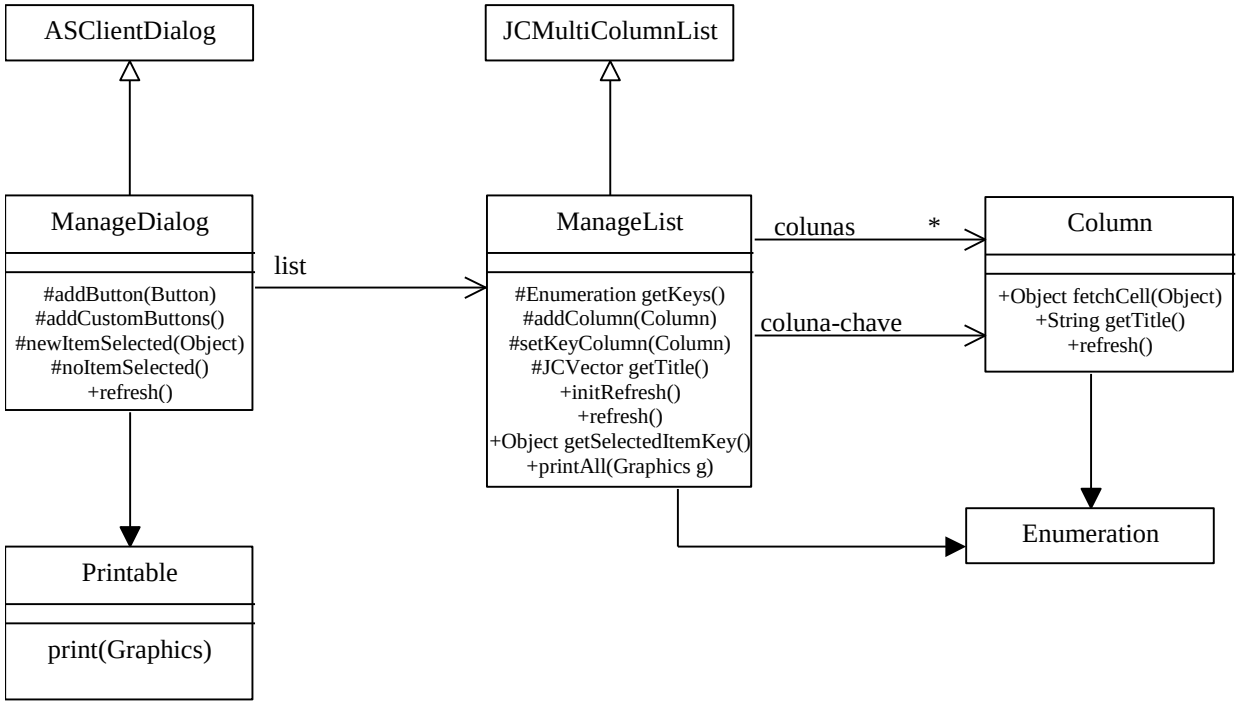

Figura 11: Classes genéricas das janelas de gestão de listas de itens – diagrama de classes

Como exemplo de uma classe que estende `ManageDialog` temos `PropertiesDialog`. Esta classe define uma janela que serve para visualizar/alterar as propriedades de um agente. Esta classe precisa de definir o método `addCustomButtons()` de forma a especificar quais os botões que estão presentes nesta janela. A introdução destes botões é feita invocando o método `addButton()`. Além disso precisa redefinir os métodos `newItemSelected()` e `noItemSelected()`, de forma a que certos botões fiquem activos/desactivos consoante um item esteja seleccionado ou não.

A Figura 11 ilustra o diagrama de classes das janelas de gestão de listas de itens enquanto que a Figura 12 e a Figura 13 ilustram, respectivamente, a janela de gestão de propriedades de um agente e o diagrama de classes respectivo.

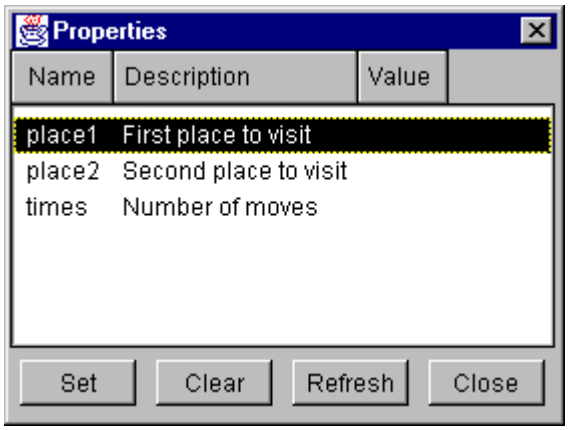

Figura 12: Janela de gestão de propriedades de um agente

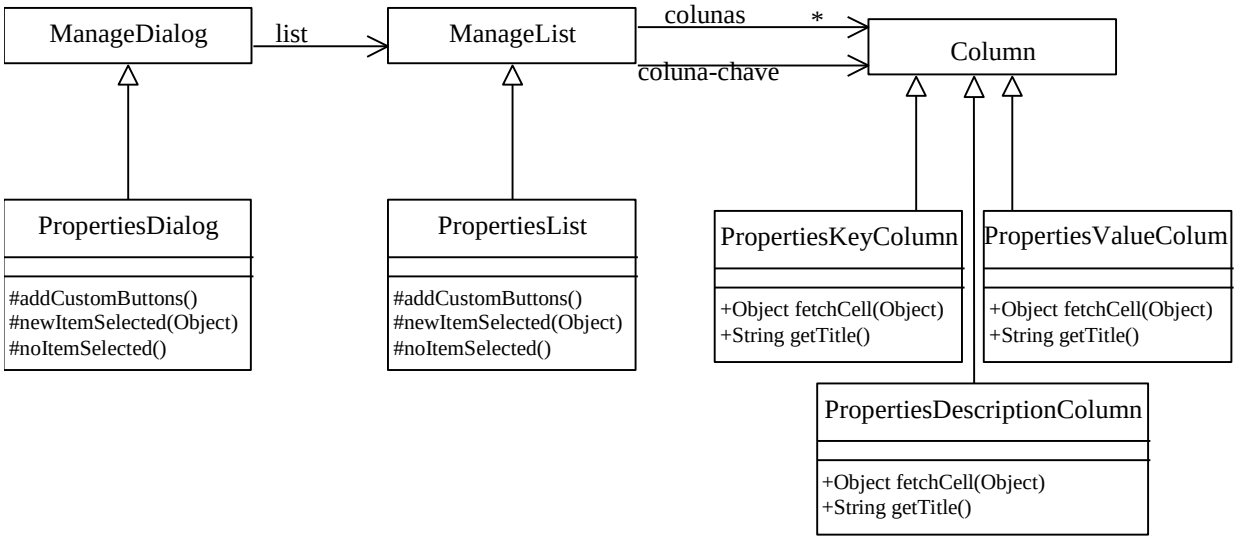

Figura 13: Janela de gestão de propriedades de agentes – diagrama de classes

3.4.4 Gestão das janelas interface de agente

Estas janelas providenciam um método de comunicação com o agente, servindo assim para iniciar , ou configurar um agente. A classe `InterfaceException` define excepções relacionadas com essas janelas, como por exemplo, a janela de inicialização do agente não existir, estando no entanto definida na meta classe do agente. A classe `InterfaceListener` serve para um utilizador fechar a janela de interface, mesmo que esta não disponha de nenhum botão que feche a mesma.

Figura 14: Janela de interface do agente PingPong.

Um exemplo de uma janela de interface de agente é a que se mostra na Figura 14 a qual ilustra a interface do agente PingPong.

3.4.5 Gestor de janelas

O gestor de janelas definido pela classe `WindowManager` mantém informação actualizada sobre o estado das janelas filhas da aplicação Cliente-AS. Deste modo a operação "close" ou "print" do menu referem-se sempre à janela filha activa. A interface `ActiveWindowListener` serve para um objecto ser notificado quando há, ou não janelas filhas activas, e qual , dessas janelas está activa.

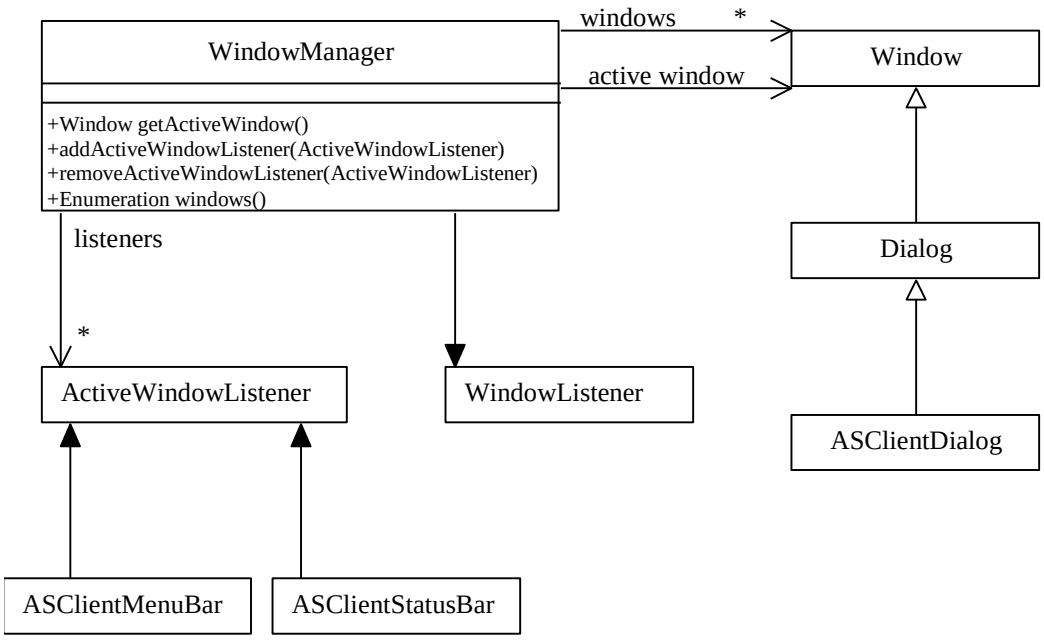

Figura 15: Gestor de janelas – diagrama de classes

De modo a manter informação actualizada sobre as janelas, `WindowManager` implementa `WindowListener`. Deste modo é notificado dos eventos que as janelas originam. O diagrama de classes é apresentado na Figura 15.

3.4.6 Excepções

A aplicação Cliente-AS, sendo robusta, é capaz de identificar situações anómalas, mesmo que estes tenham origem num ambiente externo à aplicação, recuperando sempre que possível dessas situações.
Estas situações, podem dar origem a excepções ou a erros. Os erros têm normalmente uma natureza irrecuperável, como falta de memória por exemplo, e não são tratados pela aplicação. As excepções, pelo contrário, são normalmente recuperáveis.

As excepções em Java são definidas pela classe `Exception`. Excepções especificas devem estender essa classe.
O Cliente-AS pode tratar vários tipos de excepções com origens em várias camadas:

- **Ao nível do Cliente-AS:** Estas excepções são originadas pela própria aplicação cliente, sendo essa excepção, representada por uma subclasse da classe abstracta `ASClientException`. As subclasses são:

  - **`InterfaceException:`** Esta classe define as excepções que são originadas quando se verificam inconsistências em alguma janela de interface de agente.

  - **`AccessException:`** Representa as excepções originadas pelos mecanismos de segurança do Cliente-AS. Por exemplo quando um utilizador tenta efectuar uma operação à qual não tem a permissão respectiva.

  - **`ConnectionException:`** Operações que envolvem comunicação com um Servidor-AS, tal como a visualização dos agentes disponíveis ou a modificação das permissões de um grupo, só podem ser efectuadas se existir uma ligação activa. Como as opções do menu que permitem efectuar essas operações normalmente estão desactivadas quando não existe uma ligação, esta excepção, poderá nunca ter origem, mas existe por motivos de robustez.

- **Ao nível do Servidor-AS e API-AS:** Estas excepções são normalmente instâncias de `AgentSpaceException` e são relativamente gerais.

- **Ao nível do ORB:** estas excepções são normalmente instâncias de `VoyagerException`. Esta camada também pode dar origem a erros não recuperáveis, como é o caso do "internal error" o qual pode ser originado pelo *classloader* de um objecto que tenha sido inicializado em modo *applet*.

De referir ainda que excepções ao nível da linguagem como `NullPointerException` podem ser possíveis no caso de existir algum erro na programação do Cliente-AS.

3.5 Notificação de eventos

Este mecanismo foi introduzido com a finalidade de permitir que as janelas de interface dos agentes possam ser notificadas dos eventos lançados pelo agente respectivo. Porém este mecanismo é suficientemente genérico para permitir que qualquer objecto possa receber eventos de um agente, bastando que esse se registe no agente, como ouvinte desses eventos através do método `addAgentListener()` e implemente a interface `AgentListener()`. Os eventos recebidos são subclasses de `AgentEvent`.

A transmissão destes eventos é feita através de uma classe mediadora chamada `VagentEventReceiver`. Esta classe, processada pelo vcc, permite que os agentes possam enviar eventos a objectos que não se encontrem na programa do agente. Depois de receber um evento, essa classe mediadora envia-o ao objecto destinatário.

Este mecanismo pode facilmente ser estendido, de forma a incorporar eventos de sistema de agente, bem como eventos de locais de execução e contexto. Este assunto é discutido em detalhe em **Desenvolvimentos futuros** na secção **Conclusões**.

4 Resultados

4.1 Cliente-AS

Na introdução refere-se que o objectivo principal deste trabalho consiste no desenvolvimento de uma aplicação cliente que permita de forma simples, controlada e segura, a gestão e monitorização remota de servidores AgentSpace, e Aplicações Baseadas em Agentes.

Esta aplicação, denominada Cliente-AS, foi posteriormente testada pela aplicação CELIA, uma ABA de comércio electrónico de livros baseada na Internet/Web de âmbito aberto. Esta aplicação serviu para demonstrar a eficácia da resolução do objectivo referido.

Seguidamente enumeram-se os resultados:

- **Gestão e monitorização remota de servidores AgentSpace:** Este objectivo foi atingido, superando-se a limitação imposta pela maioria dos *browsers* de limitar a comunicação do *applet* ao servidor Web que o serve, conseguindo-se assim, estabelecer ligações com qualquer Servidor-AS. Uma limitação do Cliente-AS prende-se com o facto do seu desempenho não ser elevado. Devido ao facto de não ser usado nenhum mecanismo de notificação no Cliente-AS verifica-se, por vezes, inconsistências entre a informação mantida no Cliente-AS e Servidor-AS, sendo a actualização desta informação feita de forma explicita pelo utilizador. Formas de resolução dos dois problemas referidos são abordados em detalhe em **Desenvolvimentos futuros** na secção **Conclusões**.

- **Gestão e monitorização remota de ABA:** Este objectivo também foi atingido, sendo essa gestão efectuada quer de forma estática, utilizando apenas os mecanismos existentes no Servidor-AS e Cliente-AS, usando para isso as janelas que mostram informação sobre os agentes disponíveis, ou agentes de um determinado utilizador por exemplo, quer de forma dinâmica, utilizando para isso operações definidas pelas entidades responsáveis pelo desenvolvimento das ABA, e que podem ser acedidas através das janelas interface dos agentes.

- **Gestão simples, controlada e segura:** A simplicidade foi conseguida, por um lado, através do uso de uma interface gráfica intuitiva, e por outro, pelo modelo seguido na construção dos menus ser orientado ao perfil de utilizador. A gestão de recursos controlada e segura foi conseguida através da activação/desactivação das opções existentes consoante as permissões de cada utilizador, bem como através de mecanismos de identificação de utilizador desencadeados nas operações mais críticas, tal como a alteração da palavra-chave de um utilizador.

Uma característica do Cliente-AS é inexistência de imagens e simplicidade da sua interface. Esta característica deve-se ao factor desempenho, já que a utilização de imagens, por exemplo nos botões, aumentava o tamanho do arquivo transmitido no

carregamento do *applet*, ou aumentava a interacção com o servidor Web quando os botões com essas imagens aparecessem.

4.2 Estudo de Aplicações Baseadas em Agentes

4.2.1 Estudo de SSA

Para verificar as características e importância das ABA, procedeu-se ao estudo de vários sistemas de suporte de agentes móveis. Desses sistemas destacam-se: Odyssey da General Magic [GM97]; Aglets da IBM [IBM97]; Voyager da ObjectSpace [Obj97a]; e AgentSpace do INESC [SMD98]. De referir que esses sistemas estavam ainda em desenvolvimento, tendo incidido o estudo em versões Beta e Alfa desses sistemas.

4.2.1.1 Odyssey 1.0 Beta 2

O Odyssey é um produto da General Magic que nasceu da crescente popularidade do Java para o desenvolvimento de aplicações distribuídas e do sistema de agentes pioneiro mais divulgado mundialmente, o Telescript da General Magic. O Odyssey é um conjunto de bibliotecas de classes em java que suporta o desenvolvimento de aplicações móveis distribuídas.

4.2.1.2 Aglets Workbench Alpha 5

O AWB (Aglets Workbench) é um sistema de agentes móveis baseado em Java desenvolvido pela IBM. O seu nome provém de uma mistura entre *Agent* e *Applet*, originando o *Aglet*, o objecto móvel capaz de se mover de uma máquina para outra dentro da rede. Numa determinada altura um *aglet* que se está a executar numa máquina pode terminar a sua execução, fazer a migração para uma máquina remota e recomeçar a sua execução. Quando o *aglet* se desloca ele leva consigo o seu código como o estado de todos os objectos que manipula. Este sistema dispõe de um mecanismo de segurança que lida com *aglets* potencialmente suspeitos e dispõe ainda de um meio de comunicação que permite aos agentes comunicarem síncrona ou assincronamente. Ainda de realçar a GUI (Grafical User Interface) amigável do servidor de agentes que permite facilmente manipular agentes através de operações como *Create*, *Dispatch*, *Dispose*, *Clone*, *Retract*. Ainda nessa interface é possível configurar várias opções como: preferências gerais para customização do servidor, preferências de rede para afinação de HTTP e de Web Proxies e preferências de segurança.

4.2.1.3 Estudo comparativo: Odyssey vs Aglets

De forma a compreender as características dos sistemas de agentes procedemos ao estudo comparativo entre Odyssey e AWT, desenvolvendo agentes que explorassem as funcionalidades que se pretendia estudar. Como resultado foi elaborado um

relatório comparativo das várias características dos dois sistemas que apresentamos no relatório intercalar. A Tabela 1 apresenta um resumo das características destes dois sistemas.

	Odyssey 1.0 Beta 2	**Aglets Workbench Alpha 5**
Agente	Agent Worker: Estende Agent e permite a especificação de um lista de tarefas a efectuar (uma por local de execução)	Aglet: referenciado externamente através de uma interface denominada AgletProxy
Contexto de Execução	Place: local de execução de agentes, possivelmente hierárquico. Bootplace: Estende Place e é responsável pelas operações de inicialização	AgletContext: Fornece uma interface de ambiente de execução em que se encontra o agente.
Criação de Agentes	Instanciar a classe do agente e chamar o método start. Não se pode criar agentes remotamente.	Chamar o método createAglet de AgletContext, ou clonar o agente através do método clone. Existe a possibilidade de criar um agente remotamente.
Destruição de Agentes	Quando a instância de BootPlace é destruída.	Usar o método dispose() da classe Aglet.
Mensagens	Esta funcionalidade não é suportada.	Mensagens síncronas (Now-Type) Mensagens assíncronas (Future-Type e OneWay-Type)
Persistência de Agentes	Não é suportada	Suportada, chamando o método deactivate da classe Aglet.
Segurança	Não inclui nenhum gestor de segurança	AgletSecurityManager Um agente tem 2 niveis de segurança: trusted ou untrusted
Transporte	RMI, DCOM, ou IIOP RMI necessário para o mecanismo de Finder	ATP (Agent Transfer Protocol) modelado no protocolo HTTP
Outras caracteristicas	AuditTrail Colaboração de Agentes Publicação de Objectos	Modelo de Eventos

Tabela 1: Resumo comparativo entre Odyssey e Aglets

4.2.1.4 Voyager

Quanto ao Voyager, este seria melhor classificado como infra-estrutura de comunicação do que como SSA, sendo mais válida uma comparação com o RMI da Sun [Obj97c] ou outro ORB, estando essa comparação fora do âmbito do nosso trabalho. Esta afirmação prende-se com o facto de os seus "agentes" não manterem qualquer informação relativa ao seu local de execução nativo, nem ao seu utilizador o que contradiz a definição de agente adoptada [S98].

4.2.2 CELIA: uma ABA de comércio electrónico

Uma vez que o Aglets é o sistema de agentes que apresenta mais funcionalidades, é escolhido como modelo de referência para comparação com o AgentSpace. Para estes dois sistemas deve ser desenvolvida uma ABA denominada CELIA, que tem por finalidade:

1. Testar as funcionalidades bem como o desempenho do sistema de agentes AgentSpace.

2. Demonstrar a necessidade e utilidade do paradigma de computação baseado em agentes, nomeadamente no que respeita à mobilidade. Para um maior aprofundamento sobre a utilidade deste paradigma consultar [S98].

3. Verificar o funcionamento e eficácia do Cliente-AS como mecanismo de gestão de ABA.

CELIA é uma aplicação baseada em agentes de comércio electrónico de livros baseada na Internet/Web de âmbito aberto. Esta aplicação suporta a existência de pelo menos três tipos distintos de entidades: clientes, livreiros e mediadores.

O sistema CELIA suporta o aparecimento e desaparecimento de quaisquer das entidades referidas de forma dinâmica e fácil. Clientes e livreiros actuam de forma a averiguarem a existência de determinado livro. Os mediadores providenciam serviços de páginas amarelas (de livreiros e eventualmente de livros).

Foi desenvolvida ainda uma última entidade, um simulador, para correr e testar a aplicação. Este simulador apresenta uma janela de interface ao utilizador para configuração de parâmetros de teste da aplicação tais como: número de livrarias e de livros, local de execução do cliente e locais das livrarias.

Nesta aplicação baseada em agentes, cada utilizador interactua com o seu agente especializado definindo-lhe e delegando-lhe tarefas mais ou menos complexas. Por outro lado, os agentes interactuam entre si de forma a realizarem as tarefas que lhe foram delegadas. A Figura 16 ilustra uma visão esquemática e de alto nível da aplicação.

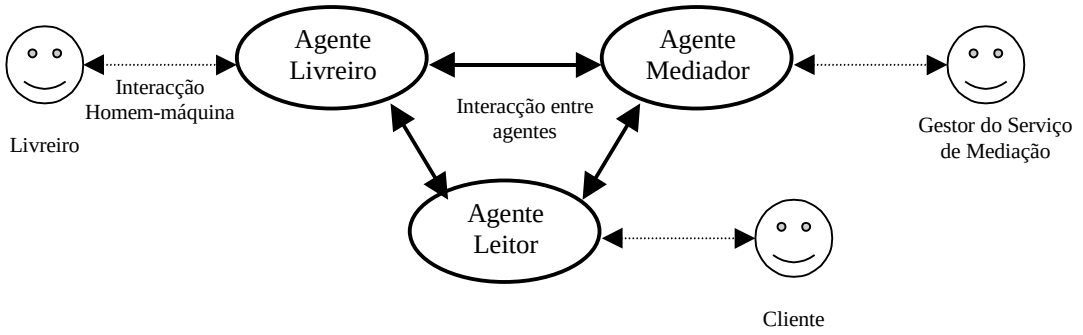

Figura 16: Interacção entre agentes e utilizadores ou agentes.

De referir que outras aproximações podiam ser utilizadas para desenvolver a aplicação. Para comparar diferentes (abordagens) aproximações de desenvolvimento de uma aplicação deste género consultar [S98].

4.2.2.1 Estrutura da aplicação

Foram definidas três classes de agentes para representar os três principais intervenientes da aplicação: o cliente, o livreiro, e o mediador. Definiu-se ainda uma classe que funciona como simulador da execução da aplicação, que cria e controla os restantes agentes, bem como contabiliza os seus respectivos tempos de execução. As quatro classes principais de agentes são:

- **`TimeTest`:** Classe do agente simulador. Cria, controla e monitoriza os tempos de execução dos restantes agentes.

- **`Broker`:** Classe do agente mediador. Mantém listas de livreiros e respectivos livros, recebe e trata pedidos quer de agentes livreiros quer de agentes clientes.

- **`BookStore`:** Classe do agente livreiro. Mantém listas de livros com os respectivos preços. Pode registar-se/desregistar-se em um ou mais agentes mediadores. Responde a pedidos de clientes directamente.

- **`Cliente`:** Classe do agente leitor. Este agente procura um livreiro que ofereça o melhor preço de um determinado livro. Comunica inicialmente com o agente mediador de forma a obter a lista dos livreiros que supostamente têm o livro pretendido (depende da versão da aplicação). Em seguida interactua com todos os agentes livreiros que recebeu do mediador para averiguar qual o que apresenta o melhor preço. A interacção entre estes agentes e os agentes livreiros varia consoante a versão realizada.

4.2.2.2 Funcionamento da Aplicação

A aplicação funciona lançando o agente simulador. Este desencadeia todo o resto da execução da aplicação e contabiliza os tempos de execução dos agentes cliente nas suas mais variadas implementações (ver as diferentes versões implementadas).

A Figura 17 ilustra, através de um diagrama de sequências UML, o cenário típico e abstracto das interacções entre os agentes envolvidos.

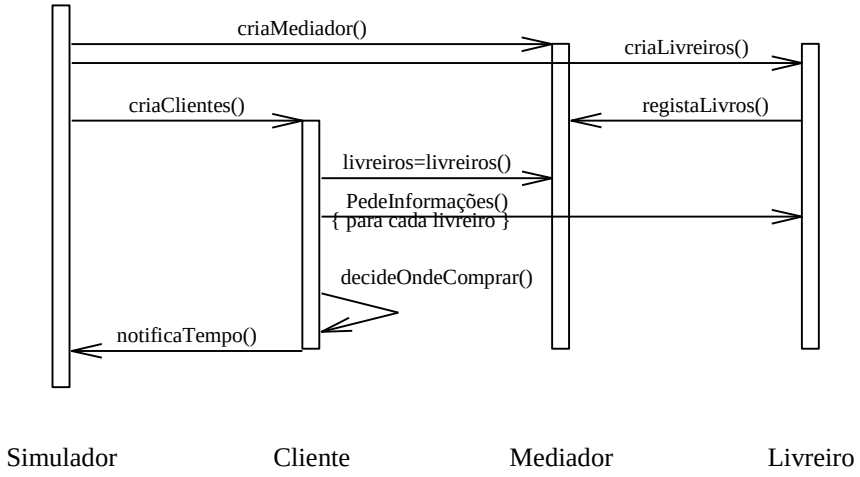

Figura 17: Principais interacções entre os agentes da aplicação CELIA.

Inicialmente o simulador cria os agentes mediadores e livreiros. Os livreiros, aquando da sua criação, enviam uma mensagem com o seu nome, os livros disponíveis e a sua identificação ao agente mediador que guarda essa informação. Depois de todas as livrarias estarem criadas e registadas no mediador, o agente simulador começa então a criar os agentes clientes que vão averiguar o melhor preço para o livro que pretendem. Os agentes cliente pedem ao agente mediador a lista de todos os livreiros registados (consoante a versão…). Quando o agente cliente termina a sua pesquisa notifica o tempo ao simulador que contabiliza os tempos de execução dos agentes cliente.

4.2.2.3 Testes Comparativos e Versões

Para estudos comparativos foram desenvolvidas duas versões da aplicação com objectivos complementares:

- Uma versão com objectivos de análise de desempenho de distintas semânticas de comunicação.

- Outra versão, com o objectivo de análise de desempenho de uma aproximação de comunicação remota face a uma aproximação de comunicação local precedida por operações de navegação.

Para cada versão definiram-se duas situações distintas:

- Execução de todos os agentes na mesma máquina.

- Execução dos agentes em máquinas distintas.

4.2.2.4 Semânticas de Comunicação

Esta primeira versão foi desenvolvida para testar o desempenho dos agentes cliente utilizando diferentes semânticas de comunicação, nomeadamente: síncrona, síncrona diferida e assíncrona.

Na Figura 18 encontra-se esquematizado o funcionamento dos clientes. Inicialmente o cliente contacta o agente mediador para obter a lista de livrarias registadas que possuem o livro pretendido. De seguida, deve contactar cada uma das livrarias com o objectivo de obter o preço desse mesmo livro.

No caso da semântica síncrona o cliente fica bloqueado até receber a resposta, passando de seguida à próxima livraria.
Na síncrona diferida o cliente continua a contactar livrarias mantendo uma referência para o resultado de cada livraria contactada. Quando acabar de contactar todas as livrarias bloqueia-se à espera dos resultados.

Na assíncrona, tal como na diferida, o cliente continua a contactar livrarias, cabendo às livrarias a tarefa de contactar o cliente enviando o resultado.

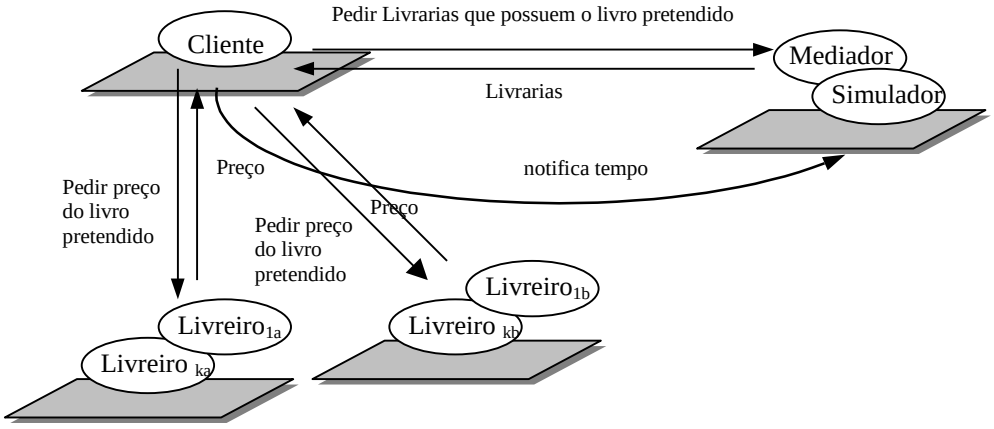

Figura 18: Funcionamento dos clientes

Para os testes desta versão o agente simulador cria o mediador e um número especificado de livrarias. Seguidamente, cria e contabiliza os tempos de execução dos clientes. Os clientes são criados um a um de forma a que num determinado instante apenas um cliente se possa executar. Cada livraria tem apenas 2 livros, sendo um destes o livro que o cliente procura. Nestes testes variou-se o número de livreiros.

4.2.2.5 Comunicação vs Navegação

Nesta segunda versão do protótipo CELIA analisou-se o desempenho relativo entre interacções remotas e interacções locais (precedidas de operações de navegação). O objectivo é confirmar se as operações de navegação podem ou não ajudar em situações de troca intensa de informação (maior volume de informação trocada) entre agentes separados geograficamente.

Com este propósito, modificou-se ligeiramente o modelo de interacção referido anteriormente, na primeira versão. Nesta segunda versão não é relevante variar o número de livrarias, mas sim o número de livros que cada uma possui. Nesta versão, a entidade mediadora não tem informação acerca dos livros que cada livraria possui, mas sim apenas a identificação de todas as livrarias registadas. Outra modificação consiste na remoção do serviço de consulta do preço de um livro por parte das livrarias, sendo substituído por um serviço que devolve todos os livros existentes bem como o seu preço, ficando assim a tarefa de filtragem do livro pretendido destinada ao cliente. Estas modificações permitem assim variar o volume de informação trocada entre clientes, livrarias e mediador, por forma a ir de encontro com os objectivos propostos. São comparadas duas abordagens:

- **Comunicação remota:** O cliente (ClientMsg) interactua remotamente e segundo uma semântica síncrona (por ser a mais eficiente) com os agentes livreiros. Saliente-se que agora o cliente solicita e recebe a lista de todos os livros existentes em cada livreiro.

- **Comunicação local precedida por operação de navegação:** O cliente (ClientMov) move-se para o local de execução do mediador, no qual obtém a lista de livreiros existentes (vid. Figura 19). A seguir cria um agente estafeta passando-lhe a lista de livrarias conhecidas e o livro a procurar (vid. Figura 20). Este então percorre todas as livrarias, movendo-se para os respectivos locais e aí realizando a pesquisa e filtragem necessária, mantendo o valor do melhor preço encontrado bem como a identificação da livraria correspondente. O estafeta comunica localmente, segundo uma semântica síncrona, com os diferentes livreiros. Quando a visita a todos os livreiros termina, o estafeta retorna ao local onde foi originalmente criado e notifica o resultado ao seu respectivo cliente. Na próxima sequência de figuras apresenta-se esquematizado o funcionamento desta abordagem.

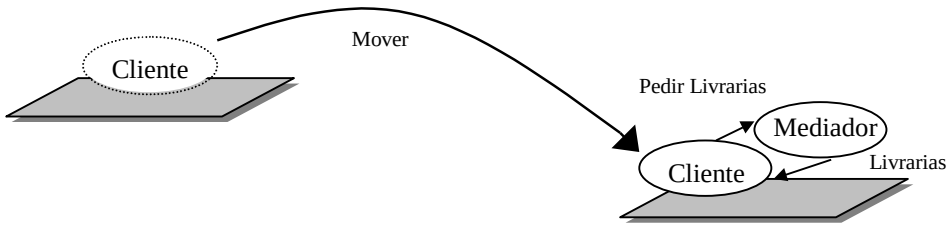

Figura 19: Obtenção das livrarias por parte do cliente junto do mediador.

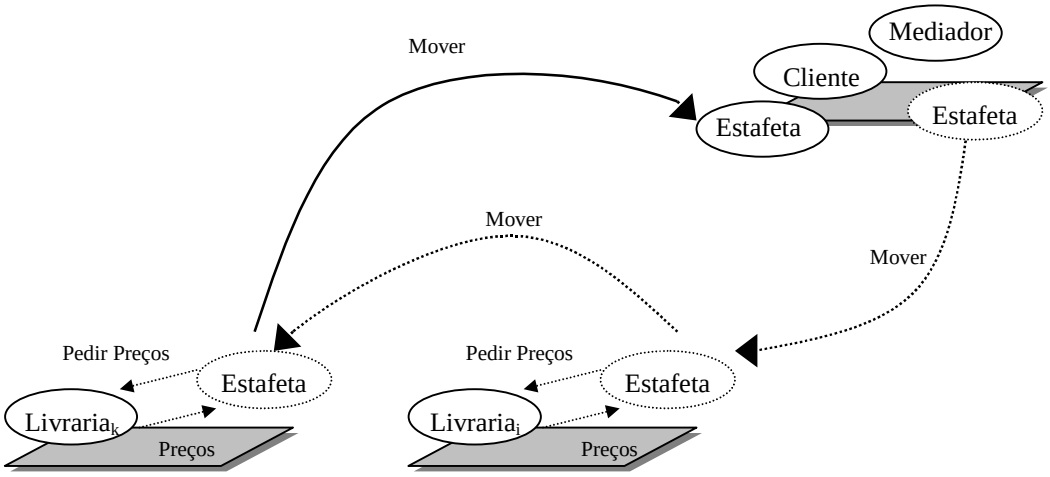

Figura 20: Funcionamento do estafeta.

Neste teste o agente simulador cria o mediador bem como cinco livrarias. Cada livraria tem um número especificado de livros, com excepção de uma livraria que tem mais um livro do que as outras, e que corresponde ao livro que o cliente procura.

Seguidamente, o simulador cria e contabiliza os tempos de execução dos clientes. Os clientes são criados um a um de forma a que num determinado instante apenas um cliente se possa executar.

4.2.3 Resultados

As análises de desempenho baseiam-se nos tempos de execução dos agentes clientes. Estes agentes quando terminam as suas tarefas (método `beforeDie` no AgentSpace ou `onDisposing` no Aglets) notificam o agente simulador dos tempos de execução das tarefas. Os testes foram realizados em três contextos diferentes: Na mesma máquina, em máquinas diferentes na mesma rede local de computadores (LAN), e em máquinas de domínios diferentes (apenas no caso do AgentSpace). Para os testes, foram usadas máquinas com processador Pentium com 16 e 32 Mb de RAM, bem como um i486DX2 com 16 Mb de RAM, sobre o JDK 1.1 para ambiente Windows 95. Foi ainda utilizada uma máquina UNIX nos testes de diferentes domínios, sendo esta a única máquina não pertencente à LAN referida.

Os valores apresentados estão em milisegundos e correspondem ao tempo médio de execução de 5 clientes. No **Apêndice C - Semânticas de Comunicação**, apresenta-se a comparação de desempenho entre clientes que utilizam mensagens remotas do tipo Síncrono, Síncrono Diferido e Assíncrono. No **Apêndice D - Navegação vs. Comunicação**, apresenta-se a comparação de desempenho entre clientes que utilizam navegação face aos que utilizam comunicação remota para as suas interacções.

4.2.3.1 Análise de desempenho

Da análise dos tempos conclui-se que, na situação de máquinas diferentes – sejam em diferentes domínios ou no mesmo, os clientes que usam mobilidade são mais lentos do que os que usam mensagens remotas para um número reduzido de livros (pouca interactividade). No entanto, os tempos dos clientes que usam mobilidade não aumentam significativamente, comparativamente aos que usam mensagens remotas, com o aumento do número de livros. Deste modo, verifica-se que existe um número de livros a partir do qual os clientes que usam mobilidade executam-se mais rapidamente do que os restantes.

Na situação da mesma máquina, os clientes que utilizam mobilidade executam-se mais lentamente para um número reduzido de livros, parecendo que, o aumento do tempo de execução com o aumentar de livros, é pouco significativo, pelo menos se comparado com o aumento descrito na situação anterior para o cliente que usa mensagens remotas. Desta forma, pelo menos, para o número de livros testados, o Cliente que usa mensagens remotas é mais rápido. Este facto advém das mensagens transmitidas na mesma máquina não serem serializadas.

Comparando os resultados obtidos no Aglets aos obtidos no AgentSpace, verifica-se que, na situação da mesma máquina no Aglets o cliente que utiliza mobilidade é mais lento do que o seu equivalente no AgentSpace, verificando-se o contrário para o cliente que usa mensagens remotas. Quanto à situação de máquinas diferentes, o cliente do Aglets que usa mobilidade é mais rápido do que o do AgentSpace, verificando-se a situação contrária para o cliente que usa mensagens remotas.

É de salientar ainda que no teste em máquinas diferentes no mesmo domínio os clientes que utilizam mensagens remotas são prejudicados devido ao facto de a máquina destinada aos clientes ser a mais lenta. No entanto, se optássemos por colocar as livrarias nessa máquina, acabaríamos por prejudicar os escravos que lá se deslocassem, bem como o cliente que tem de esperar pelo resultado desse escravo. Se colocássemos o mediador nessa máquina, também prejudicávamos o cliente que usa mobilidade, porque este desloca-se à máquina do mediador. Mesmo assim a conclusão que pelo menos a partir de um certo número de livros, a mobilidade é mais rápida, deverá manter-se.

Na situação de diferentes domínios, sendo uma das máquinas que suporta as livrarias, uma máquina multi-utilizador, os valores podem ter sido, em certo grau, influenciados pela carga da máquina que possivelmente não se manteve constante ao longo dos testes.

4.2.3.2 Diferenças na implementação

A implementação da aplicação CELIA foi concebida de forma que o código fosse tão semelhante quanto possível – designadamente para que fossem credíveis os resultados comparativos de desempenho.

No entanto, há a realçar as seguintes diferenças, que em geral correspondem a benefícios do AgentSpace relativamente ao *Aglets*, segundo a perspectiva do programador:

- **Manutenção de canais abertos**: O Aglets, contrariamente ao AgentSpace não possui esta característica, ou seja um agente para comunicar com outro agente tem de conhecer, para além da sua identificação, também a sua localização corrente. Como consequência desta particularidade, o exemplo CELIA só funciona no Aglets, ao admitir-se certos pressupostos, tal como o facto do agente mediador não se mover.

  Benefício para o programador: **Transparência de localidade, abstracção, simplicidade**.

- **Obtenção de uma referência através da identificação do agente**: No AgentSpace é possível obter uma referência (`AgentView`) para um agente a partir da identificação do agente denominada `AgentID`. Em Aglets para se obter tal referência (`AgletProxy`), além de fornecer a identificação do agente (designada por `AgletID`), é também necessário fornecer um URL, que corresponde ao local (`AgletContext`) onde o agente se encontra. (A classe `AgentInfo`, definida para a versão Aglets foi necessária devido a este motivo). Adicionalmente esta referência no AgentSpace é mais poderosa que a homóloga no Aglets, já que esconde a localização do agente e providencia de facto uma política de controlo de acessos.

  Benefício para o programador: **Simplicidade, transparência**.

- **Possibilidade de especificar o método a executar depois de uma operação de navegação** - Contráriamente ao AgentSpace onde é possível a especificação do método que vai ser executado após a conclusão de uma operação de navegação, no Aglets é sempre executada a *callback* `onArrival`. (Devido a esta característica foi necessário introduzir na versão Aglets do escravo ou do cliente que usa mobilidade, um campo que mantém o estado do agente. Assim no `onArrival` invocamos o método desejado com base no teste a esse estado.

  Benefício para o programador: **Elegância, simplicidade, manutenção**.

- **Possibilidade de enviar mensagens contendo objectos arbitrários** – No AgentSpace é possível enviar uma mensagem contendo um objecto cuja classe não existe na máquina onde o receptor da mensagem se encontra. No Aglets as mensagens remotas não causam transferência de *bytecode*, e assim as classes usadas na mensagem têm de estar instaladas nas máquinas envolvidas. Na aplicação CELIA esse facto traduziu-se na instalação das classes usadas nas mensagens nas diferentes máquinas.

  Benefício para o programador: **Abstracção, manutenção**.

5 Conclusões

5.1 Conclusões

A partir dos resultados obtidos verifica-se que o Cliente-AS cumpre os objectivos propostos, sendo de facto um meio de gerir remotamente um ou mais SSA e ABA.

Uma limitação do Cliente-AS prende-se com o facto do seu desempenho não ser elevado, como referido nos resultados. Para esta limitação foi sugerida uma solução apresentada em **Desenvolvimentos futuros**.

Verificou-se também que, por vezes, surgem inconsistências entre a informação mantida no Cliente-AS e Servidor-AS, sendo a actualização desta informação feita de forma explicita pelo utilizador. Para este problema também foi sugerida uma solução que se apresenta em **Desenvolvimentos futuros**.

Na introdução referiu-se que a importância deste trabalho estava relacionada com a importância das ABA e do SSA AgentSpace. O estudo apresentado em **Resultados** verifica essa importância, incidindo esse estudo sobre ABA que tirem partido da mobilidade. Os resultados desse estudo permitem concluir que ABA que usem mobilidade são de facto mais eficientes se o grau de interacção entre os agentes envolvidos for elevado. Além disso verifica-se que as ABA são flexíveis, como se demonstra pela utilização de vários agentes cliente com comportamentos distintos, permitindo assim que cada interveniente possa desenvolver os seus próprios agentes.

Verificou-se também que o *applet* é um mecanismo excelente de integração com a Internet/Web, permitindo a execução de aplicações com um certo grau de complexidade.

5.2 Desenvolvimentos futuros

5.2.1 Extensão do mecanismo de notificação de eventos

Como já foi referido, uma funcionalidade introduzida no Servidor-AS e API-AS é um mecanismo de eventos, que permite a um objecto registar-se como ouvinte de eventos de um ou vários agentes. Este mecanismo tem a finalidade de permitir que as janelas interface de agente actualizassem automaticamente o seu conteúdo com base em eventos lançados explicitamente pelos agentes. Estes eventos são definidos pela entidade que desenvolve o agente.

Um desenvolvimento futuro seria o desenvolvimento de classes de eventos de sistema, os quais seriam lançados automaticamente pelo agente em resposta a operações como remoção desse agente ou alteração das suas propriedades. O Cliente-AS deveria registar-se nesse agente de modo a receber esses eventos. Um mecanismo de eventos semelhante deveria ser desenvolvido para os locais de execução e

contexto, permitindo assim ao Cliente-AS receber notificações dos locais, tal como a chegada de um agente visitante ou, no caso do contexto, ser notificado da introdução de uma meta classe de agente.

Vantagens desta aproximação:

- **Eliminação da inconsistência entre a informação do Cliente-AS e Servidor-AS:** Como o Cliente-AS é notificado das alterações de estado ocorridas no Servidor-AS esta incoerência deixa de existir.

- **Aumento da simplicidade da interface utilizador:** Esta vantagem deriva da eliminação do botão "refresh" que perde a sua utilidade como meio explicito de actualizar a informação mantida no Cliente-AS sobre o Servidor-AS.

Desvantagens:

- **Perca de eficiência no Servidor-AS e Cliente-AS:** Esta perca de eficiência resulta do facto de a notificação de eventos consumir recursos. No entanto este problema é minimizado já que o tipo de mensagens usadas pela notificação de eventos é assíncrono por omissão.

- **Diminuição do factor de escalabilidade do Servidor-AS:** Resultante da desvantagem anterior. Poderia ser resolvido atribuindo uma prioridade mais baixa à actividade de notificação de eventos ou desenvolvendo um tampão de eventos. Este tampão acumularia os eventos um a um, existindo uma actividade responsável por enviar o conjunto dos eventos acumulados nesse tampão aos mediadores dos objectos que estão registados como ouvintes desses eventos. Essa actividade deveria estar inactiva um certo período de tempo *t* antes de voltar a enviar outro conjunto de eventos. O mediador deveria, após receber esse conjunto de eventos, enviá-los uma a um e ordenadamente, ao objecto ouvinte desses eventos. A Figura 21 ilustra, através de um diagrama de sequências UML, a notificação de eventos de agentes, num cenário típico usando o tipo de notificação por omissão, que é assíncrono.

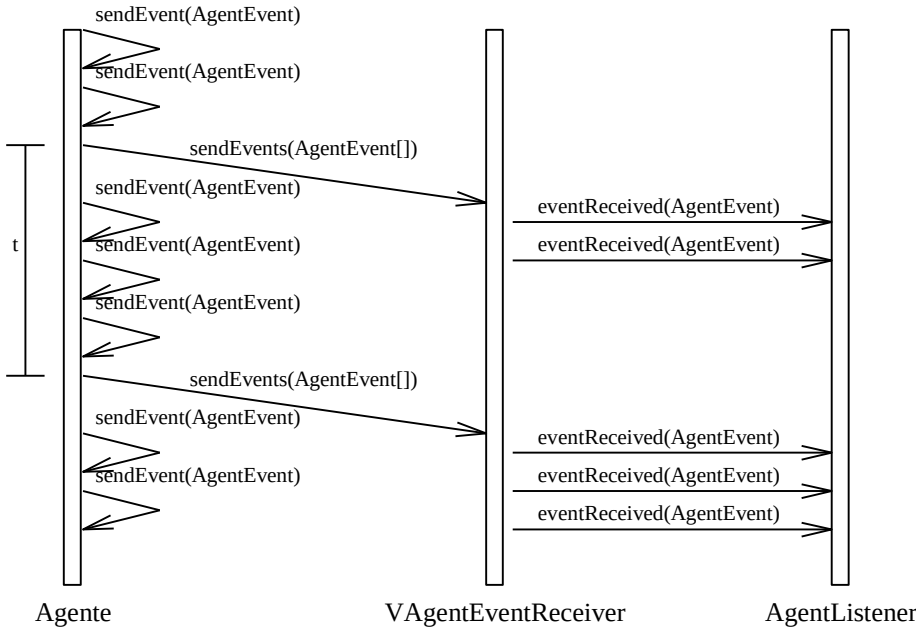

Figura 21: Notificação de eventos de agente usando um tampão - diagrama de sequências UML

5.2.2 Cache

Como possível trabalho futuro, e de forma a atenuar os problemas de desempenho do Cliente-AS, sugerimos que as interacções entre este e o Servidor-AS sejam reduzidas ao mínimo possível. Para tal sugerimos o desenvolvimento de uma cache para o Cliente-AS.

Devido à grande diversidade de informação acessível, seria vantajoso subdividir essa cache em várias sub-caches, cada qual especialista no tipo de informação que gere.

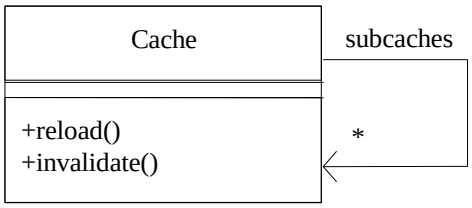

Figura 22: Possível cache genérica do Cliente-AS – diagrama de classes

A Figura 22 ilustra uma possível concepção para essas caches. Assim uma cache é um objecto com possíveis sub-caches que também são caches. Um objecto do tipo cache deve implementar a interface `Cache`.

Qualquer interacção com o servidor deve assim ser mediada por objectos do tipo `Cache`. Assim se quando quisermos obter a lista de meta classes de agente

disponíveis num determinado contexto, não deverá haver qualquer interacção com o Servidor-AS desse contexto a menos que esta informação não se encontre em cache.

O método `reload()` deve ir buscar informação ao Servidor-AS de modo a preencher essa cache, enquanto que `invalidate()` deve invalidar o seu conteúdo. O método `reload()` é adequado, para operações de refrescamento de informação explícito como a invocada pelo botão "refresh", ou implícito como a remoção ou criação de um agente.

A classe `Connection`, que como já foi referido é responsável por gerir as ligações deveria implementar a interface Cache e ter referências às outras sub-caches disponibilizando-as aos objectos que necessitem de efectuar operações sobre objectos do mesmo tipo daquele que é gerido por essas caches.

Como exemplo de utilização destes objectos no Cliente-AS vamos considerar a operação de visualização/configuração das propriedades de um agente. A Figura 23 ilustra o diagrama de classes envolvidas nesta operação. Para efectuar esta operação, vamos precisar de:

- **Obter uma lista de agentes:** Sejam estes os agentes disponíveis, ou de agentes cujo dono é o utilizador corrente. Esta lista será obtida através de uma instância de `AvailableAgentsCache` ou `MyAgentsCache` as quais estendem `AgentsCache`.

- **Obter informação relativa a um agente:** A gestão dessa informação deverá ser feita por uma instância de `AgentCache`.

- **Obter a lista de propriedades de um agente:** Através de uma instância de `PropertiesCache`

- **Obter informação sobre uma propriedade:** Uma instância de `PropertyCache` deverá guardar o nome, valor e descrição de uma propriedade.

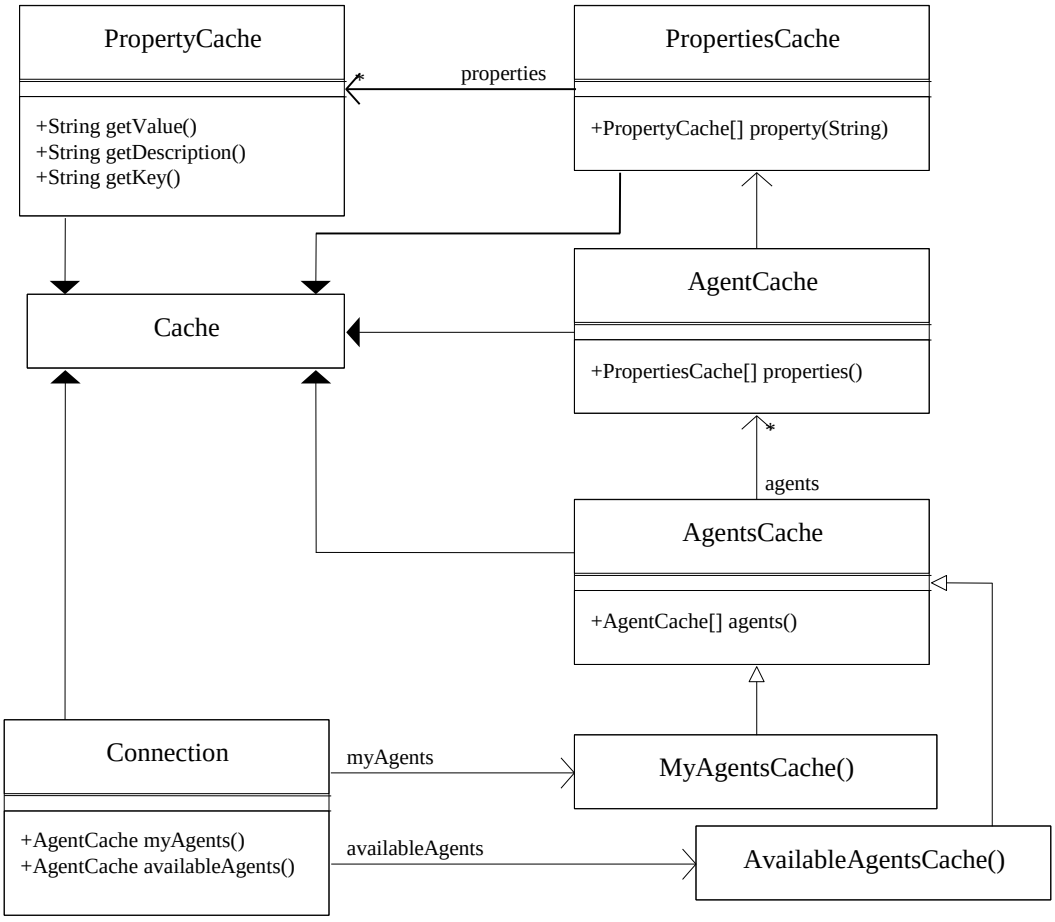

Figura 23: Cache relativa às propriedades de um agente – diagrama de classes

Vantagens desta aproximação:

- **Aumento de desempenho:** Relacionado com a diminuição da interacção entre o Cliente-AS e o Servidor-AS.

- **Facilidade de gerir a comunicação:** Devido principalmente à centralização da comunicação com o Servidor-AS nas caches, o que permite uma gestão mais fácil dessa comunicação.

- **Facilidade de incorporar a notificação de eventos no Cliente-AS:** Este mecanismo foi descrito anteriormente e ajusta-se perfeitamente ao modelo de cache descrito, bastando que cada cache implemente a interface que permite receber eventos dos objectos que correspondem ao tipo de informação gerida por essa cache.

Desvantagens:

- **Aumento do tempo de carregamento do applet:** Derivado do aumento do número de classes que compõem o Cliente-AS, aumentando assim o tamanho do arquivo que é carregado da página *html*.

- **Aumento da inconsistência entre a informação do Cliente-AS e Servidor-AS:** Este problema que já se verificava, é aumentado se quisermos tirar benefícios da cache e diminuir a frequência do refrescamento da informação contida na cache. No entanto este problema pode ser resolvido através do mecanismo de eventos descrito anteriormente.

6 Referências

[GM97] General Magic, Inc. *Odyssey Product Information*. 1997. http://www.genmagic.com/technology/odyssey.html

[IBM97] IBM Tokyo Research Laboratory. *The Aglets Workbench: Programming Mobile Agents in Java*. 1997. http://www.trl.ibm.co.jp/aglets

[MD94] Microsoft Corp., Digital Corp. *Common Object Model Specification*, Draft ver. 0.2, Outubro 1994.

[Obj97a] ObjectSpace. *Voyager Core Technical User Guide*. Version 1.0. 1997.

[Obj97b] ObjectSpace. *Voyager And Agent Platforms Comparision*. 1997.

[Obj97c] ObjectSpace. *Voyager And RMI Comparision*. Version 1.0. 1997.

[OMG91] OMG. *The Common Object Request Broker: Architecture and Specification (CORBA)*. 1991.

[P92] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. (3ª edição), McGraw-Hill. 1992

[R96] Jonh Rodley. *Writing Java Applets*. 1996. Coriolis Group Books. http://www.coriolis.com

[S98] Alberto Silva, *Espaço de Agentes: Suporte, Desenvolvimento e Gestão de Aplicações Baseadas em Agentes, Dinâmicas e Distribuídas*, IST/UTL, Tese de Doutoramento, 1998.

[SMD97] Alberto Silva, Miguel Mira da Silva, José Delgado. *A Survey of Web Information Systems*. Proceedings of the Conferência WebNet'97 – World Conference of the WWW, Internet & Intranet, Novembro 1997.

[SMD98] Alberto Silva, Miguel Mira da Silva, José Delgado. *AgentSpace: An Implementation of a Next-Generation Mobile Agent System*. (Mobile Agents'98) Lecture Notes in Computer Science, 1477, Springer Verlag, Setembro 1998.

[Sun98a] Sun Microsystems, Inc., *JavaSoft Home Page*, 1998. http://www.javasoft.com/

[Sun98b] Sun Microsystems, Inc., *The Java Development Kit (JDK)*. 1998. http://www.javasoft.com/products/jdk/

[Sun98c] Sun Microsystems, Inc., JavaSoft. *Java Remote Method Invocation (RMI)*, 1998. http://www.javasoft.com/products/jdk/rmi/

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley Publishing, 1991.