DIOGO DE CAMPOS
MAURÍCIO OLIVEIRA HAENSCH

UMA PLATAFORMA PARA DESENVOLVIMENTO DE SISTEMAS MULTIAGENTE BDI NA WEB

Florianópolis

Julho de 2011

DIOGO DE CAMPOS
MAURÍCIO OLIVEIRA HAENSCH

UMA PLATAFORMA PARA DESENVOLVIMENTO DE SISTEMAS MULTIAGENTE BDI NA WEB

Monografia apresentada na Universidade Federal de Santa Catarina para obtenção do título de Bacharel em Ciências da Computação.

Orientador:

Prof◦ Dr. Ricardo Azambuja Silveira

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Florianópolis

Julho de 2011

Título: Uma plataforma para desenvolvimento de sistemas Multiagente BDI na Web

Autores: Diogo de Campos, Maurício Oliveira Haensch

Banca Examinadora:

Prof. Dr. Ricardo Azambuja Silveira
Universidade Federal de Santa Catarina - UFSC

Profa. Dra. Jerusa Marchi
Universidade Federal de Santa Catarina - UFSC

Prof. Dr. Jomi Fred Hübner
Universidade Federal de Santa Catarina - UFSC

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

.nexe	*Native Client executable*,	p. 37
Java ME	*Java Platform, Micro Edition*,	p. 24
Java SE	*Java Platform, Stantard Edition*,	p. 24
3APL	*Artificial Autonomous Agent Programming Language*,	p. 21
3APL-M	*Artificial Autonomous Agent Programming Language - Mobile*,	p. 24
API	*Application Programming Interface*,	p. 38
BDI	*Beliefs-Desires-Intentions*,	p. 7
dMARS	*distributed Multi-Agent Reasoning System*,	p. 19
FIPA	*Foundation for Intelligent Physical Agents*,	p. 28
GDB	*GNU Debugger*,	p. 37
HTML	HyperText Markup Language,	p. 34
IA	Inteligência Artificial,	p. 6
IAD	Inteligência Artificial Distribuída,	p. 6
IDE	*Integrated Development Environment*,	p. 40
IMDb	*The Internet Movie Database*,	p. 32
JADE	*Java Agent DEvelopment Framework*,	p. 28
NaCl	*Google Native Client*,	p. 36
PGP	*Partial Global Planning*,	p. 17
PRS	*Procedural Reasoning System*,	p. 19
SDK	*Software Development Kit*,	p. 37
SMA	Sistemas Multiagente,	p. 10

SUMÁRIO

1 INTRODUÇÃO

A aplicação da Inteligência Artificial (IA) em tarefas do cotidiano torna-se cada vez mais comum, fazendo parte em situações das mais simples às mais complexas. A IA engloba uma variedade de subcampos, sendo utilizada para diversas atividades intelectuais humanas (RUSSELL; NORVIG, 2003).

Cada problema a ser resolvido pode empregar diferentes técnicas da IA, de acordo com suas características. Essa variabilidade propicia o avanço de várias subáreas da Inteligência Artificial, das quais uma das mais recentes é o estudo de Sistemas Multiagente (SMA). Este ramo da Inteligência Artificial Distribuída (IAD) propõe a utilização de um conjunto de agentes autônomos inteligentes para buscar a solução de um problema. Por ser uma técnica mais recente, ainda existe muito espaço para realizar estudos sobre ela.

Devido ao constante aumento da utilização de computação e também da Internet para realização e automatização de atividades rotineiras, a aplicação da IA torna-se também cada vez mais presente e surgem possibilidades para diversos usos de técnicas de IA, nos mais variados problemas, sendo o uso de técnicas de IA cada vez mais comum e transparente ao usuário. O objetivo do presente trabalho é explorar os SMA e sua utilização em aplicações voltadas para a Web.

1.1 MOTIVAÇÃO

Sistemas Multiagente é uma das técnicas recentes da IA que pode ser integrada em diversas situações, e sua utilização na Web, para variados fins, não é diferente. Porém, esse tipo de técnica, por suas características, ainda não é tão utilizada quanto outras técnicas mais simples e consolidadas da IA.

As possibilidades de aplicação de agentes autônomos são várias. Já existem vários agentes na Internet, em tarefas como providenciar serviços facilitando o uso de interfaces, permitindo maior segurança e dinamismo no *e-commerce*, e também agentes jogadores, que ajudam (ou até substituem) jogadores humanos em jogos online (SHOHAM; LEYTON-BROWN, 2009).

Apesar de agentes já estarem sendo utilizados para aplicações específicas, a implementação

desses agentes não é tão simples e não existem metodologias e ferramentas consolidadas para o desenvolvimento de agentes, que acaba por dificultar a expansão da área na Web.

Com a criação de uma plataforma para desenvolvimento de agentes deliberativos, esperamos contribuir para uma maior utilização de agentes inteligentes em diversas aplicações voltadas para a Web, procurando realizar o processamento no lado do cliente, para permitir programas mais complexos sem sobrecarregar o servidor.

Neste trabalho iremos definir a situação em que se encontram os sistemas multiagentes na Web hoje, com o intuito de encontrar e especificar as dificuldades existentes na criação dos mesmos. Com estas informações em mãos, iremos pesquisar soluções existentes em ferramentas, linguagens e metodologias usadas no desenvolvimento de agentes, tanto na Web quanto fora dela. Baseado nestas pesquisas, será proposta uma nova plataforma para criação de sistemas multiagentes utilizando o modelo de agentes *Beliefs-Desires-Intentions* (BDI) para utilização na Web.

1.2 OBJETIVO GERAL

O objetivo deste trabalho é propor uma ferramenta para o desenvolvimento de aplicações com sistemas multiagente na Web.

1.2.1 OBJETIVOS ESPECÍFICOS

1. Identificar e compreender as principais ferramentas e aplicações que utilizam Sistemas Multiagente já existentes na Web;
2. Estudar plataformas de desenvolvimento de agentes;
3. Propor uma plataforma de desenvolvimento utilizando uma linguagem adequada para descrição de agentes BDI;
4. Implementação da plataforma proposta;
5. Desenvolver exemplos de aplicações utilizando a plataforma;
6. Realizar testes, analisar e validar o projeto;

2 INTELIGÊNCIA ARTIFICIAL E AGENTES

Atualmente, a IA abrange uma enorme variedade de subcampos, desde áreas de uso geral, como aprendizado e percepção, até tarefas específicas como jogos de xadrez, demonstração de teoremas matemáticos, criação de poesia e diagnóstico de doenças. A IA sistematiza e automatiza tarefas intelectuais e, portanto, é potencialmente relevante para qualquer esfera da atividade intelectual humana. (RUSSELL; NORVIG, 2003).

Desde o surgimento do termo Inteligência Artificial dentro da computação, várias áreas que estudam o raciocínio humano vem evoluindo juntas, como a Neurociência, Filosofia e Psicologia. O reflexo, dentro da Computação, da evolução conjunta dessas e de outras ciências, é o surgimento de novas técnicas e o aprimoramento das já existentes, tentando compreender ao máximo a mente humana e representá-la computacionalmente.

A IA se preocupa não apenas em compreender, mas também em construir entidades inteligentes (RUSSELL; NORVIG, 2003). Com isso, a ideia de agentes autônomos como entidades inteligentes se torna mais natural, graças à similaridade com a inteligência humana, onde é possível criar ambientes complexos a partir da interação entre diversos agentes inteligentes individuais.

Agentes podem ser empregados não apenas em aplicações complexas, mas também em tarefas simples e rotineiras, podendo ser considerados uma nova categoria de ferramenta na sociedade informatizada atual (BRENNER; WITTIG; ZARNEKOW, 1998). Tarefas cotidianas na Web podem ser desempenhadas por um agente de software ao invés de um usuário humano, como, por exemplo, um programa que verifica determinados sites de comércio eletrônico diariamente por promoções de um determinado produto, e avisa ao seu usuário ao encontrar uma boa oferta. Por outro lado, ao se utilizar mais de um agente para uma dada atividade, a complexidade da aplicação já aumenta consideravelmente, visto que novos aspectos são incluídos, como a comunicação entre eles. Essa outra categoria de aplicações é objeto de estudo de um sub-ramo da IA, chamado de IAD.

2.1 INTELIGÊNCIA ARTIFICIAL DISTRIBUÍDA

Alguns dos ramos em evolução da computação, desde o surgimento dos computadores, são a distribuição e a paralelização do processamento. Um sistema distribuído pode ser caracterizado como um sistema onde componentes, tanto de hardware quanto software, se comunicam e coordenam suas ações através de mensagens (COULOURIS; DOLLIMORE, 1988). A subárea da computação que estuda sistemas distribuídos, suas características e aplicações, é denominada Computação Distribuída. O que incentiva a pesquisa nessa área é a possibilidade de dividir um problema grande em pequenas partes para que seja resolvido distribuidamente. Isso possibilita que a solução seja alcançada de forma mais rápida e eficiente, além de aumentar a confiabilidade e permitir uma maior expansibilidade do sistema.

Assim como várias áreas da Computação, a IA também encontra espaço na Computação Distribuída, onde é chamada de Inteligência Artificial Distribuída. A IAD busca exatamente a criação de sistemas distribuídos inteligentes, envolvendo aspectos de projeto e interação entre esses sistemas, estratégias de solução de problemas, mecanismos de comunicação e cooperação e diversas outras questões presentes em sistemas distribuídos.

Segundo a divisão ilustrada na figura 1, proposta por (GASSER; BOND, 1988) e muito utilizada, a IAD possui três grandes áreas. A Inteligência Artificial Paralela busca mais eficiência através da divisão de um sistema inteligente em processos menores e sua paralelização, adicionando recursos e reduzindo o tempo total de processamento. A área de Resolução de Problemas Distribuídos, por outro lado, não busca dividir um sistema já pronto a nível de processos, por questões de performance, mas sim decompor um problema em módulos que cooperam e trocam conhecimento para alcançar a solução do problema (BRENNER; WITTIG; ZARNEKOW, 1998).

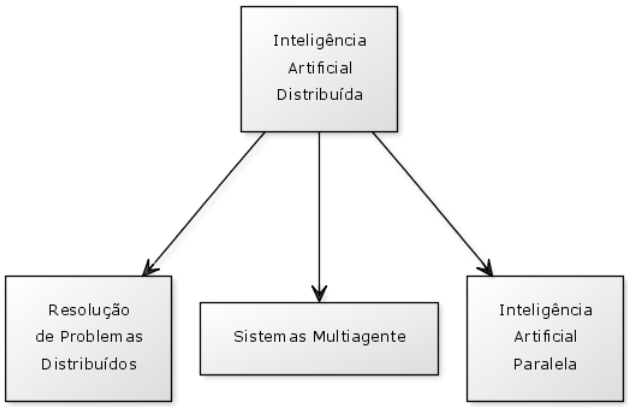

Figura 1: Ramos da Inteligência Artificial Distribuída. Fonte: BRENNER; WITTIG; ZARNEKOW; 1998.

Outra característica da Resolução de Problemas Distribuídos é o conhecimento prévio do problema a ser resolvido. O sistema é projetado baseado no problema a ser solucionado, e essa é a principal diferença com relação aos Sistemas Multiagente (SMA). Os agentes integrantes de um SMA muitas vezes são desenvolvidos sem ter como preocupação principal alcançar a melhor solução do problema geral. É comum que agentes sejam desenvolvidos e depois integrados ao sistema, e, consequentemente, outros aspectos são incorporados a esses sistemas. Essa terceira divisão da IAD será mais discutida adiante, na seção de Sistemas Multiagente.

2.2 AGENTES

Um agente pode ser definido como uma entidade computacional que possui um comportamento autônomo, permitindo-lhe decidir suas próprias ações (ALVARES; SICHMAN, 1997). Esse comportamento autônomo inclui a percepção do ambiente, através de sensores, e suas ações sobre ele, por meio de seus atuadores (RUSSELL; NORVIG, 2003). Além disso, a definição de agente é bem ampla, e inclui tantos agentes físicos, em *hardware*, quanto agentes puramente em *software*. Para ambos os casos, uma série de características são comuns aos agentes, mesmo que em diferentes níveis:

- **Reatividade**: capacidade de perceber o ambiente e realizar uma ação de acordo com o estímulo recebido;
- **Autonomia**: o agente pode tomar suas próprias decisões de acordo com os estímulos do ambiente e seu conhecimento prévio;

- **Cooperação**: capacidade de cooperar com outros agentes para atingir um objetivo em comum;
- **Comunicação de conhecimento**: capacidade de se comunicar com outros agentes para troca de conhecimento e informações, em alguma linguagem de mais alto nível, e não apenas uma comunicação seguindo algum protocolo;
- **Personalidade**: demonstração de uma personalidade do agente;
- **Adaptabilidade**: habilidade de se adaptar a mudanças no ambiente em que atua;
- **Capacidade de inferência**: capacidade de criar conclusões a partir de informações do ambiente e conhecimento prévio;
- **Mobilidade**: possibilidade de migrar para uma outra plataforma;
- **Continuidade temporal**: continuidade do agente com suas características mesmo após longos períodos de tempo;

Além dessas características e da classificação em agentes em *hardware* e *software*, eles também são comumente classificados em duas categorias: reativos e deliberativos, também chamados cognitivos. Agentes reativos são mais simples, programados para reagir a estímulos do ambiente e de outros agentes, enquanto agentes deliberativos são baseados em algum ciclo de raciocínio para tomada de decisão sobre suas ações.

Um agente reativo simples pode ser descrito através de um ciclo com poucas etapas, iniciando com a percepção do ambiente atual, complementamente desligado dos passos anteriores, visto que não existe uma memória nesse tipo de agentes. Após utilizar seus sensores para descobrir o estado do ambiente, o agente descobre qual ação deve tomar através de um conjunto de regras pré-definidas no modelo desse agente, basicamente ativando uma certa ação caso uma determinada característica no ambiente tenha sido detectada. O passo final é utilizar os atuadores para realizar as ações devidas, e iniciar esse ciclo novamente (RUSSELL; NORVIG, 2003).

A inspiração para agentes reativos são modelos de organização biológica mais simples, onde um indivíduo sozinho não possui características inteligentes porém uma comunidade de agentes apresenta um comportamento visivelmente inteligente, como o caso de formigas (agentes reativos com pouca ou nenhuma inteligência) e um formigueiro (uma comunidade inteligente) (BITTENCOURT, 2001). Modelos reativos de agente funcionam com base em estímulos e respostas, sem possuir uma memória, planejar ações futuras ou se comunicar com outros agentes. Todo o conhecimento de um agente sobre a ação dos demais é percebida através de modificações no ambiente.

Por outro lado, agentes cognitivos possuem inspiração em organizações mais complexas, como o caso de organizações sociais humanas como grupos, hierarquias, mercados. Esses agentes possuem uma representação interna explícita do ambiente, de outros agentes e uma memória que é a base do planejamento de suas ações futuras. Além disso, agentes cognitivos também possuem um sistema de comunicação distinto do de percepção, diferentemente dos agentes reativos. Outra característica que distingue sistemas feitos por agentes reativos e sistemas de agentes cognitivos é a quantidade de agentes envolvidos, visto que no caso de agentes reativos um sistema pode contar com uma população na ordem de milhares de membros enquanto sistemas de agentes cognitivos costumam contar com um número reduzido, na ordem de dezenas de entidades (BITTENCOURT, 2001).

Um dos modelos existentes de agentes cognitivos é o modelo BDI, que pode ser considerado um modelo deliberativo mas procura manter a reatividade de um agente. A ideia desse modelo é manter as características de deliberação, possuindo uma representação interna do ambiente e memória de suas ações, mas sem perder a capacidade de reagir rapidamente a eventos. Mais informações sobre esse modelo podem ser vistas a seguir.

2.2.1 MODELO *BELIEFS-DESIRES-INTENTIONS*

O modelo BDI representa uma arquitetura cognitiva de agentes e é inspirado no conceito de estados mentais, com origem em estudos do raciocínio prático humano, na teoria de mesmo nome do filósofo Michael Bratman (BRATMAN, 1999). A representação de uma teoria do raciocínio prático humano pela IA é uma tentativa de se criar entidades ainda mais inteligentes e autônomas. Além de uma representação de um modelo interno do ambiente, característica que difere agentes deliberativos dos reativos, agentes seguindo a arquitetura BDI também possuem a capacidade de tomar decisões lógicas, através de seu raciocínio. Porém, agentes BDI foram modelados para responderem a estímulos enquanto deliberam, procurando aliar o raciocínio a longo prazo com reações rápidas a mudanças no ambiente.

Um agente BDI utiliza seu conhecimento, representado por seu modelo interno, como parte do seu processo de raciocínio que acaba por alterar seu estado, também referido como estado mental. Esse estado é composto pelos três conceitos propostos pela teoria de Bratman, crenças, desejos e intenções, assim como dois novos conceitos utilizados em sistemas que seguem esse modelo: objetivos e planos. Esses cinco componentes definem o chamado estado mental do agente BDI e podem ser caracterizadas como a seguir (BRENNER; WITTIG; ZARNEKOW, 1998):

Crenças (*Beliefs*): representam o conhecimento do agente sobre o mundo, e não necessaria-

mente são informações corretas, podendo ser incompletas em muitos casos.

Desejos (*Desires*): estão relacionados a estados do mundo que o agente gostaria de provocar. Podem existir desejos conflitantes entre si, ou com poucas chances de se tornar realidade, e influenciam na tomada de decisão sobre a próxima ação a tomar.

Objetivos (*Goals*): formam um subconjunto dos desejos onde o agente tem possibilidade de atuar. Por esse motivo, diferentemente dos desejos, não devem existir objetivos contraditórios, devendo ser consistentes.

Intenções (*Intentions*): podem ser definidas como objetivos que o agente se comprometeu a realizar. Usualmente, o agente não possui recursos suficientes para tentar atingir todos seus objetivos, e assim, intenções são um subconjunto dos objetivos que o agente escolhe para efetivamente tentar alcançar, de acordo com os recursos disponíveis.

Planos (*Plans*): são uma combinação das intenções. Cada intenção constitui um subplano de um plano geral de um agente, e o conjunto de todos os planos reflete as intenções do agente.

O modelo BDI surgiu no estudo do raciocínio prático humano, que difere do raciocínio teórico por envolver decisões relacionadas à ações, e não apenas relacionadas às crenças de um agente (WOOLDRIDGE, 2000). Esse raciocínio prático pode ser dividido em duas principais etapas: a primeira, deliberação, onde decidimos qual objetivo desejamos alcançar; a segunda, raciocínio meio-e-fim (BRATMAN, 1981), é onde decidimos como alcançá-lo.

O modelo de agente deliberativo proposto por Rao/Georgeff (RAO; GEORGEFF, 1995) inclui a fundamentação teórica do conceito BDI, como definições básicas da lógica e semântica envolvidas, assim como conceitos da implementação prática de sistemas BDI, e em particular o ciclo básico de um interpretador BDI, mostrado a seguir.

2.2.2 CICLO DE RACIOCÍNIO

Na implementação de agentes BDI, duas fases distintas são representadas. A primeira fase é composta pela geração de opções e em seguida uma filtragem sobre essas opções. Esses dois processos são influenciados por diversos fatores, como o estado atual do ambiente e estado interno do agente, considerando suas crenças e intenções atuais. A filtragem é responsável por escolher a melhor opção gerada, que dará origem a uma nova intenção do agente.

A segunda etapa, raciocínio meio-e-fim (BRATMAN, 1981), é responsável por gerar um plano a ser executado pelo agente tendo em vista a realização de suas intenções. Após a es-

colha do plano, ele é executado e o ciclo de raciocínio do agente passa a uma subetapa dentro da fase de raciocínio meio-e-fim, que consiste basicamente em perceber o ambiente, com as mudanças sofridas pela execução dos planos dos agentes no sistema e novos eventos externos, e uma reconsideração das intenções do agente, podendo deixar de lado algumas que se tornaram impossíveis ou que obtiveram sucesso.

O ciclo básico descrito é definido em (RAO; GEORGEFF, 1995), e segue os passos abaixo:

1. iniciar-estado
2. repetir:
   (a) gerar-opções
   (b) selecionar-opções
   (c) atualizar-intenções
   (d) executar
   (e) receber-eventos-externos
   (f) largar-objetivos-satisfeitos
   (g) largar-objetivos-impossíveis
3. fim-repetir

Muitas extensões desse ciclo básico foram criadas, procurando adicionar alguns aspectos no processo de deliberação e raciocínio dos agentes, como o planejamento de ações futuras ao ciclo atual ou levar em consideração os outros agentes, procurando escolher intenções e planos que auxiliem no trabalho coletivo dos agentes no sistema.

A utilização de agentes individuais é eficiente e já muito utilizada. Como exemplos de agentes em *hardware*, há vários tipos de robôs, com sensores e atuadores específicos para uma tarefa à qual foram projetados. Agentes implementados em software podem ser encontrados em aplicações de auxílio ao ensino e comércio eletrônico, por exemplo. Mas a utilização de sistemas com múltiplos agentes (SMA) também já é muito comum para resolução de problemas mais complexos, onde a utilização de apenas um agente não é satisfatória ou apropriada.

2.3 SISTEMAS MULTIAGENTE

A ideia de utilizar um grupo agentes é muito empregada para problemas com características de paralelismo, devido às características de autonomia dos mesmos, o que permite um alto grau

de distribuição. Como já visto, uma das divisões defendidas por (GASSER; BOND, 1988) dentro da IAD é a de Sistemas Multiagente. Essa categoria pode ser ainda separada entre Sistemas Multiagente Reativos, Sistemas Multiagente Cognitivos e Sistemas Multiagente Híbridos.

Os sistemas ditos reativos são mais simples, geralmente possuem uma quantidade de agentes na ordem de centenas ou milhares e são formados por agentes reativos. Conforme já mencionado, um único agente reativo não possui um comportamento muito inteligente, mas sistemas compostos por esses agentes podem apresentar um comportamento inteligente emergente.

Sistemas cognitivos são formados por agentes deliberativos e procuram seguir um modelo de relações humanas, deliberando sobre suas ações e não apenas reagindo. Diferentemente dos agentes reativos, agentes cognitivos possuem um sistema de comunicação, e dada a característica de raciocinarem sobre suas ações futuras, essa capacidade de comunicação é bastante explorada nesse tipo de sistemas, gerando uma implementação mais complexa com diversas novas áreas para modelagem, como gerenciamento de conflitos, negociação, cooperação e troca de informações entre agentes. Existem também sistemas híbridos, que utilizam em diferentes escalas os dois tipos de agentes, reativos e cognitivos.

Os SMA podem ser categorizados levando em conta outras características, como a divisão entre sistemas homogêneos ou heterogêneos, onde no primeiro os agentes inseridos no sistema possuem a mesma estrutura, e no segundo tipo agentes diferentes coexistem no sistema, tornando também sua implementação mais complexa. Outra divisão pode ser utilizada considerando o modelo de coordenação dos agentes do sistema, conforme figura 2:

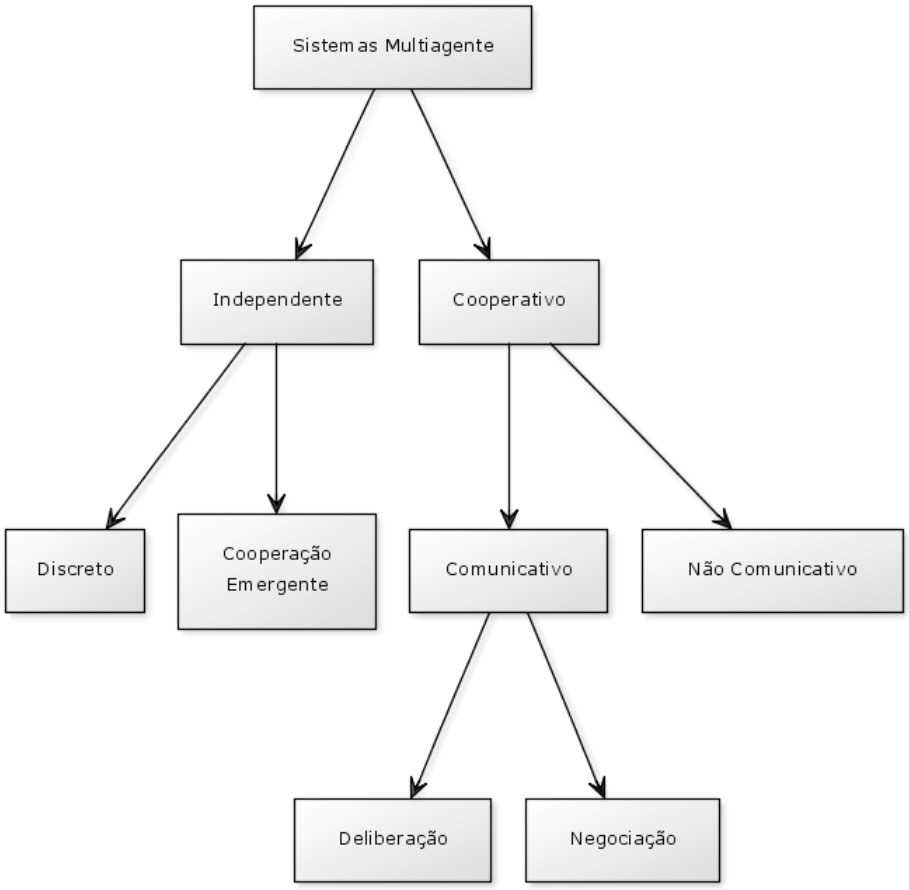

Figura 2: Modelos de coordenação de um Sistema Multiagente. Fonte: BRENNER; WITTIG; ZARNEKOW; 1998.

Conforme visto na figura, os SMA podem ser caracterizados como cooperativos ou independentes numa visão mais ampla. Dentro dos sistemas independentes, há os conceitos de sistemas discretos, onde agentes desempenham funções que não exigem uma cooperação explícita, e de cooperação emergente, que são também sistemas com agentes agindo independentemente, mas que aparentam uma cooperação para o usuário. Essa divisão fica mais clara ao se pensar em agentes desempenhando tarefas não relacionadas, no caso de sistemas discretos, e agentes que possuem um ou mais objetivos em comum, que mesmo sem efetivamente trabalharem em equipe, passam a impressão de agir em conjunto ao usuário.

Um sistema dito cooperativo possui mecanismos explícitos de cooperação entre os diversos agentes, que são criados de modo que essa comunicação os auxilie a atingir seus objetivos, fazendo com que o uso dessa habilidade de comunicação seja intencional e bastante utilizada. No ramo de sistemas cooperativos, há os sistemas não comunicativos e os comunicativos. Os não comunicativos tem como característica uma cooperação indireta entre as entidades através do ambiente, percebendo o ambiente e as mudanças cometidas nele por outros agentes, contra

uma comunicação direta no caso dos sistemas comunicativos, estes, por sua vez, podem ser subdivididos com relação ao modelo de comunicação empregado entre sistemas orientados a deliberação ou a negociação.

Sistemas com características de comunicação por deliberação seguem uma metodologia de planejamento global envolvendo todos os agentes, de modo com que todas as entidades tenham participação e decidam em conjunto o que fazer numa determinada situação. Os sistemas que utilizam negociação possuem uma característica a mais no processo de comunicação, que é a questão competitiva, além da deliberação em conjunto. Isso pode ser retratado em casos de conflitos entre entidades do sistema, e a negociação empregada pode acabar por designar tarefas a alguns agentes a fim de resolver os conflitos emergentes. Para as duas metodologias há uma série de protocolos de cooperação mais adequados, como é o caso de *Partial Global Planning* (PGP) para sistemas com deliberação e *Contract Net* para sistemas com negociação, por exemplo. Mais detalhes sobre essas metodologias podem ser vistas em (BRENNER; WITTIG; ZARNEKOW, 1998).

Há muitas outras maneiras de se categorizar SMA na literatura, visto que essa área da IA não possui, em muitos casos, definições universalmente aceitas e utilizadas. Uma outra característica que pode ser analisada, por exemplo, é a questão de mobilidade de um SMA, que pode caracterizar os sistemas entre estacionários e móveis.

Com relação aos SMA de uma forma mais ampla, existem uma série de técnicas de projeto, modelagem e implementação desses sistemas, cada uma com suas características, vantagens e desvantagens. Na seção seguinte, veremos algumas das linguagens de agente e ferramentas pesquisadas.

3 LINGUAGENS E FERRAMENTAS

A área de Sistemas Multiagente dispõe de uma série de técnicas, linguagens e ferramentas que visam organizar o desenvolvimento desse tipo de sistema. Além de linguagens e *frameworks*, existem diversas metodologias de engenharia de software orientada a agentes, que podem ser divididas entre duas vertentes principais: as que estendem metodologias tradicionais baseadas no paradigma de Orientação à Objetos, e novas metodologias, focadas diretamente em SMA.

Como uma das vantagens da primeira abordagem está a familiaridade com as metodologias já consolidadas e conhecidas, procurando atingir uma maior facilidade de uso por parte dos desenvolvedores. Mas a criação de novas metodologias é considerada por muitos uma opção com maior potencial, já que diversas características que diferem *software* tradicional de sistemas baseados em agentes podem ser levadas em consideração desde o surgimento da metodologia. Considerando as vantagens das duas opções, não há um consenso entre qual abordagem ou qual metodologia é mais adequada para o desenvolvimento de SMAs.

Além do foco em técnicas de Engenharia de Software, o universo de Sistemas Multiagente também possui diversas linguagens de programação e ferramentas para desenvolvimento de sistemas, que serão abordadas a seguir.

3.1 LINGUAGENS DE PROGRAMAÇÃO

O desenvolvimento da área de agentes é muito focado na evolução de arquiteturas, interação entre agentes e alguns outros conceitos mais teóricos. Isso acaba por limitar o crescimento do agentes na sua implementação prática (BORDINI, 2005). O objetivo das linguagens de programação orientadas a agentes, assim como o surgimento de plataformas que facilitam desenvolvê-los, é de reduzir a distância entre modelagem e implementação de agentes, permitindo uma maior facilidade de desenvolvimento de sistemas.

O termo *linguagem de programação orientada a agentes* surgiu junto com um protótipo de uma linguagem desse tipo, chamada **Agent0**, na década de 80. Essa linguagem, criada na Universidade de Stanford, acabou por influenciar as demais linguagens criadas nos anos

seguintes, até os dias de hoje. Yoav Shoham, criador dessa proposta de Programação Orientada a Agentes (POA), defende alguns pontos a favor desse conceito (BORDINI; WOOLDRIDGE; HÜBNER, 2007):

- Descrever agentes pode ser feito de uma maneira muito mais simples, utilizando uma linguagem mais familiar;
- A ideia da programação orientada a agentes é ser uma programação declarativa, onde apenas descrevemos o que precisamos, e deixamos com que o mecanismo interno conclua o que deve ser feito. Além disso, esse mecanismo interno (modelo computacional) funciona de um modo similar ao que raciocinamos, e idealmente isso permite com que programadores seguindo POA não precisem se preocupar em como um agente atinge seus objetivos.

Algumas das linguagens orientadas a agentes existentes serão analisadas a seguir:

3.1.1 AGENTSPEAK(L)

A linguagem *AgentSpeak(L)* (RAO, 1996) foi criada para o desenvolvimento de Sistemas Multiagente, e segue o modelo de agentes BDI. A criação dessa linguagem veio como uma alternativa para formalizar agentes BDI, e ela pode ser vista como uma abstração para um sistema BDI já implementado (RAO, 1996), o *Procedural Reasoning System* (PRS), que será apresentado com mais detalhes na seção seguinte, ou de uma das muitas extensões, *distributed Multi-Agent Reasoning System* (dMARS).

Programas escritos em *AgentSpeak(L)* são responsáveis por ditar o comportamento de agentes e sua interação com o ambiente. Por meio dela, um projetista pode atribuir crenças, desejos e intenções aos agentes. O conjunto de crenças de um agente descreve seu estado atual, incluindo o modelo de si mesmo, o estado do ambiente e de outros agentes. Os estados que um agente procura tornar realidade são os desejos, e as medidas adotadas para satisfazer esses desejos são suas intenções, que são escolhidas baseado em estímulos internos e externos. A alternativa adotada para unir a teoria e a prática da construção de agentes foi mudar a linguagem de programação usada para o modelo de execução de agente, de uma linguagem comum, para uma outra com um ponto de vista mais externo, orientada à descrição dos estados mentais de um agente (RAO, 1996).

Uma das metas de *design* da linguagem é encontrar um balanceamento entre a busca dos objetivos e o comportamento reativo dos agentes quando alguma situação inesperada, ou um

evento externo no ambiente ocorre durante a execução de seus planos. A linguagem procura atingir cinco pontos principais, descritos a seguir (BORDINI; WOOLDRIDGE; HÜBNER, 2007):

- Permitir delegação de tarefas ao nível de objetivos entre os agentes. Isto significa uma comunicação em um nível mais alto, descrevendo os objetivos desejados ao invés de tarefas já planejadas e calculadas.
- Oferecer suporte a solução de problemas orientada a objetivos, isto é, os agentes devem conseguir concluir as metas delegadas sistematicamente.
- Permitir a construção de sistemas que interagem com o ambiente.
- Integrar, de maneira clara, um comportamento orientado a objetivos e reativo à situações diversas.
- Permitir comunicação e cooperação em nível de conhecimento.

A definição de um agente *AgentSpeak(L)* é feita de dois componentes: o conjunto de crenças iniciais de um agente (e possivelmente objetivos iniciais) e o conjunto de planos. Um exemplo de descrição de um agente pode ser visto a seguir, na linguagem *Jason*, uma das principais implementações de *AgentSpeak*.

```
iniciado.
+iniciado <- print("Agente sendo executado").
```

A primeira linha do trecho apresentado corresponde às crenças inicias do agente. Nesse caso, o agente possui apenas uma crença que é chamada “iniciado”. A próxima parte do código apresentado demonstra os planos que o agente possui. Como foi elaborado, o agente deve imprimir a mensagem “Agente sendo executado” quando ele tiver a crença “iniciado” em seu conjunto de crenças, ou seja, a crença “iniciado” ativa o plano descrito a seguir, composto apenas de uma função já presente na linguagem (*print*).

O exemplo demonstrado é uma versão muito básica de descrição de um agente utilizando o *AgentSpeak(L)*. A linguagem possui uma série de outras funcionalidades que permitem a criação de um sistema com entidades muito mais complexas que a apresentada acima. Como algumas das características não presentes no exemplo, há a possibilidade de adicionar condições para que um plano seja executado, como também a ideia de diferentes tipos de objetivos (*achievement goal* e *test goal*, que procuram realizar ou testar um objetivo, respectivamente).

As linguagens *AgentSpeak(L)* e diversas variações dela, comumente referidas como *AgentSpeak*, causaram um grande impacto desde sua criação, influenciando diversas outras linguagens orientadas a agentes, tanto indiretamente, como fonte de inspiração, quanto sendo estendida e utilizada, como é o caso do *Jason*.

Jason[1] é um interpretador feito em *Java* de uma extensão da linguagem *AgentSpeak*. A linguagem original, apesar de ser um grande marco na programação de SMAs, é uma linguagem bastante abstrata, e, por esse motivo, surgiu essa tentativa de criar uma extensão mais prática, além de prover uma semântica operacional à linguagem (BORDINI, 2005). Algumas das características encontradas na linguagem estendida para o *Jason* são:

- Possibilidade de troca de planos entre agentes;
- Utilização de performativas para comunicação inter-agentes;
- Permitir a execução de um sistema através da rede;
- Anotações sobre a fonte das informações do modelo interno de um agente;
- Permitir diversas customizações e extensões do usuário;

O *Jason* possui uma implementação voltada para agentes Web, chamado JaCa-Web (MINOTTIE, 2010). A ideia desta ferramenta é bastante parecida com a plataforma proposta neste trabalho, mas possuem diferenças na implementeção. Mais informações sobre o JaCa-Web serão apresentadas na seção 4.2, onde falamos sobre o desenvolvimento de aplicações multiagente na Web.

3.1.2 3APL

A *Artificial Autonomous Agent Programming Language* (3APL) (DASTANI; RIEMSDIJK; MEYER, 2005) é uma outra linguagem de programação de agentes motivada por arquiteturas cognitivas que permite definir crenças, objetivos, ações, planos e regras de raciocínio. A figura 3 ilustra o funcionamento de um agente escrito nessa linguagem:

[1] O interpretador *Jason* é *Open-Source*, sob a licença GNU LGPL e disponível em http://jason.sourceforge.net

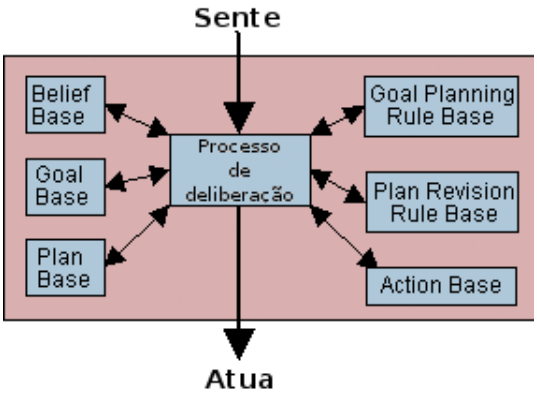

Figura 3: Organização de um agente 3APL. Fonte: BORDINI, 2005.

O projeto da linguagem levou em consideração a separação em estados mentais (estruturas de dados) e o processo de deliberação que altera esses estados (rotinas de programação) (BORDINI, 2005). Para isso, um arquivo com a descrição de um agente 3APL é estruturado da seguinte forma:

```
PROGRAM "agent"
BELIEFBASE {}
CAPABILITIES {}
GOALBASE {}
PLANBASE {}
PG-RULES {}
PR-RULES {}
```

A primeira linha indica apenas o nome do programa (agente) sendo criado. As quatro próximas linhas, *BELIEFBASE*, *CAPABILITIES*, *GOALBASE* e *PLANBASE*, possuem as estruturas que definem o estado mental de um agente e suas capacidades, indicando as crenças, ações, objetivos e planos do agente, respectivamente. As duas últimas linhas são responsáveis pela definição de regras que serão aplicadas para planejar e rever os planos. Uma descrição mais completa dessas estruturas pode ser vista abaixo (DASTANI, 2006):

- **BELIEFBASE** - Contém sentenças *Prolog*[2] (CLOCKSIN; MELLISH, 2003) (STERLING; SHAPIRO, 1994) que definem informações específicas sobre o ambiente e demais crenças do agente.

[2] *Prolog* (*PROgramming in LOGic*) é uma linguagem de programação baseada em lógica de predicados criada na década de 70. Possui várias implementações (seguindo ou não o padrão ISO definido para a linguagem).

- **CAPABILITIES** - São ações mentais que o agente pode executar e que modificam o conjunto de crenças do mesmo. Seguem a estrutura: "{Pré-condição} Nome da Ação {Pós-condição}". Desse modo, um agente que possui as crenças que satisfazem "Pré-condição" pode executar "Nome da Ação" e alterar sua base de crenças para "Pós-condição".
- **GOALBASE** - Define o conjunto de objetivos que um agente deseja alcançar.
- **PLANBASE** - Possui os planos que serão executados pelo agente para realizar seus objetivos. É possível iniciar um agente com planos já definidos, assim como gerar novos durante a execução do agente.
- **PG-RULES** - Segue a estrutura "Objetivo ← Condições | Corpo". Caso o agente tenha como meta "Objetivo" e suas crenças satisfaçam as "Condições", o "Corpo", que é um plano, é adicionado à base de planos do agente.
- **PR-RULES** - São regras de revisão de planos, e seguem uma estrutura similar à das "PG-RULES".

Além da linguagem de descrição de agentes, também foi criada a plataforma 3APL, que é uma ferramenta que permite o desenvolvimento, implementação e execução de agentes 3APL. Essa plataforma é escrita em *Java* e pode ser encontrada na página do projeto 3APL, assim como um manual de utilização. O ciclo de deliberação de um agente pode ser visto na figura 4.

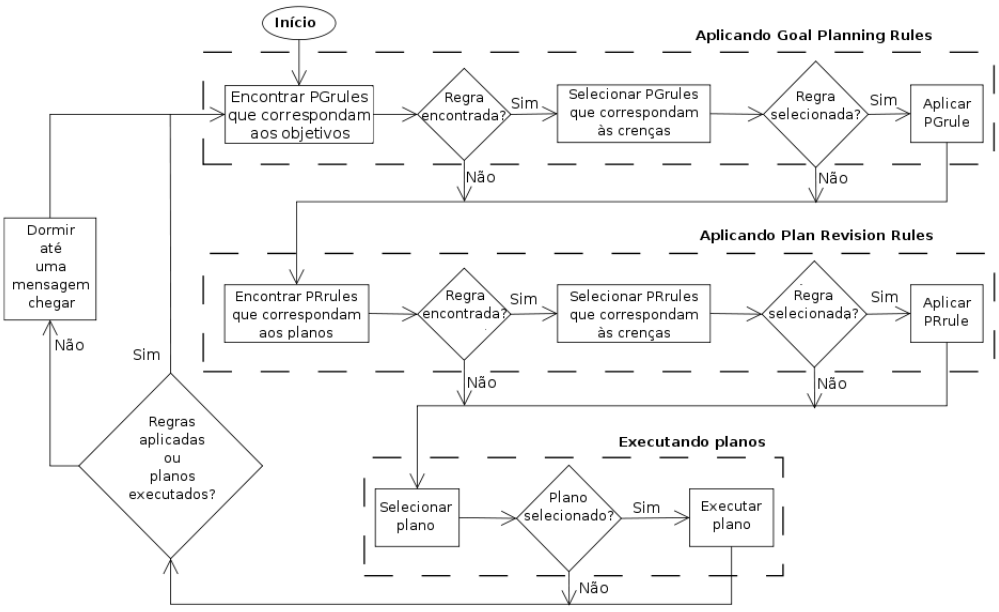

Figura 4: Ciclo de deliberação de um agente 3APL. Fonte: BORDINI, 2005.

De maneira simples, um agente 3APL passa por 3 etapas básicas: a primeira delas é a aplicação das regras de aquisição de planos baseado nos objetivos e nas crenças do agente, na segunda etapa são aplicadas as regras de revisão dos planos adquiridos, que também passam por um filtro com relação às crenças do agente, e a terceira é onde os planos são de fato executados.

Por ser escrita em *Java*, é possível imaginar uma aplicação que execute a plataforma 3APL em um *browser* através de *Applets Java*, permitindo assim a criação de sistemas multiagentes em aplicações Web, mas até o término deste trabalho, não encontramos nenhuma aplicação deste tipo.

3.1.3 3APL-M

A partir da linguagem *3APL*, foi desenvolvida uma variação para dispositivos móveis, chamada *Artificial Autonomous Agent Programming Language - Mobile* (3APL-M) (KOCH, 2005). Como os dispositivos alvo dos agentes desenvolvidos nessa linguagem em geral possuem um poder computacional mais limitado, uma das grandes características dessa linguagem é seu tamanho reduzido.

Essa nova linguagem, desenvolvida por Fernando Koch, é um subconjunto da original *3APL*, e possui interpretadores escritos em *Java* (para as versões *Java Platform, Stantard Edition* (*Java* SE) e *Java Platform, Micro Edition* (*Java* ME)) e na linguagem funcional *Haskell*[3]. A *3APL-M* possui como componentes da descrição de um agente as estruturas *CAPABILITIES*, *BELIEFBASE*, *GOALBASE* e *RULEBASE*, que funcionam de modo similar à linguagem anterior, sendo esta última estrutura a de maior diferença com relação à linguagem anterior. A figura 5 ilustra a arquitetura básica de uma máquina 3APL-M.

[3] *Haskell* é uma linguagem puramente funcional *open-source* de mais de 20 anos de pesquisa. Para mais informações sobre a linguagem visite a página oficial `http://www.haskell.org/`

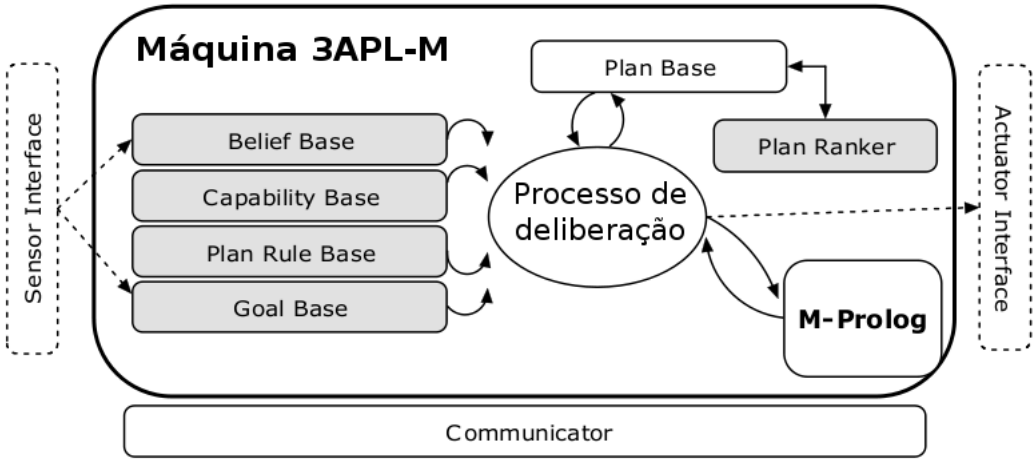

Figura 5: Arquitetura do 3APL-M. Fonte: KOCH; 2005.

Com base na figura 5, alguns dos elementos da arquitetura básica são descritos a seguir:

- Máquina 3APL-M: encapsula os componente dos Agentes e provê a API para integrá-los às aplicações *Java*;
- *Belief Base*, *Capabilities Base*, *Goal Base* e *Plan Rule Base*: São as estruturas básicas, como definidas na linguagem 3APL por (DASTANI, 2006);
- Processo de deliberação: Implementa as regras de transição lógica que criam os planos baseados nos objetivos e crenças do agente;
- *Plan Base*: A lista de planos atuais gerados pelo processo de deliberação. Esta lista pode ser manipulada por uma lógica externa, que permite a criação de analisadores de plano implementados pelo usuário;
- *Plan Ranker*: Classe interna, parte do sub-sistema de planejamento, que classifica os planos com base no seu cálculo de utilidade;
- *mProlog*: Uma versão reduzida da máquina Prolog, otimizada para aplicações móveis;
- Interfaces de Sensores e Atuadores: São as interfaces *Java* que permitem a integração da máquina 3APL-M com o mundo externo. Sensores e Atuadores são desenvolvidos em *Java* e conectados à máquina 3APL-M através de métodos especiais definidos na API da mesma;

- *Communicator*: É uma interface genérica para a infraestrutura de comunicação. Pacotes de mensagens trocados entre agentes locais são manipulados diretamente por esta infraestrutura, enquanto pacotes entre agentes externos devem ser manipulados por uma lógica externa (que depende do ambiente no qual a aplicação está sendo implementada).

Do mesmo modo que o 3APL, o 3APL-M possui uma plataforma implementada em *Java*, e portanto, poderia ser executada na Web através de uma *Applet Java*, mas não possuí uma plataforma implementada especialmente para este propósito.

O presente trabalho tem como objetivo criar uma plataforma para desenvolvimento de agentes em aplicações na Web, onde o processamento ocorrerá no lado do cliente, e não do lado do servidor, como é mais comum nas plataformas existentes. Para isso, reduzir a carga de processamento envolvida na execução dos agentes é uma estratégia muito interessante para ter melhores resultados. A linguagem *3APL-M* possui essa característica, assim como um modelo interessante de agente BDI, e, portanto, foi escolhida como base para alcançarmos nosso objetivo.

3.2 FERRAMENTAS

Já existem várias ferramentas para desenvolvimento de sistemas multiagentes, e, mesmo que não sejam voltadas para agentes web, serviram como uma grande fonte de inspiração para este trabalho. Algumas destas ferramentas serão apresentadas nesta seção.

3.2.1 PRS - *PROCEDURAL REASONING SYSTEM*

O PRS (GEORGEFF; LANSKY, 1987), é comumente citado como um das primeiras e mais completas arquiteturas para desenvolvimento de agentes BDI. Como seu nome indica, o PRS é baseado em uma representação de conhecimento procedural, que descreve como seguir uma sequência de ações para alcançar algum objetivo. Por exemplo, sabemos que a sequência de ações ”Colocar roupas na máquina de lavar, Colocar o sabão, Ligar a máquina e Esperar 45 minutos” é suficiente para se cumprir um objetivo como lavar roupas, e grande parte do conhecimento das pessoas é naturalmente procedural (HUBER, 1993).

A figura 6 ilustra o funcionamento do PRS:

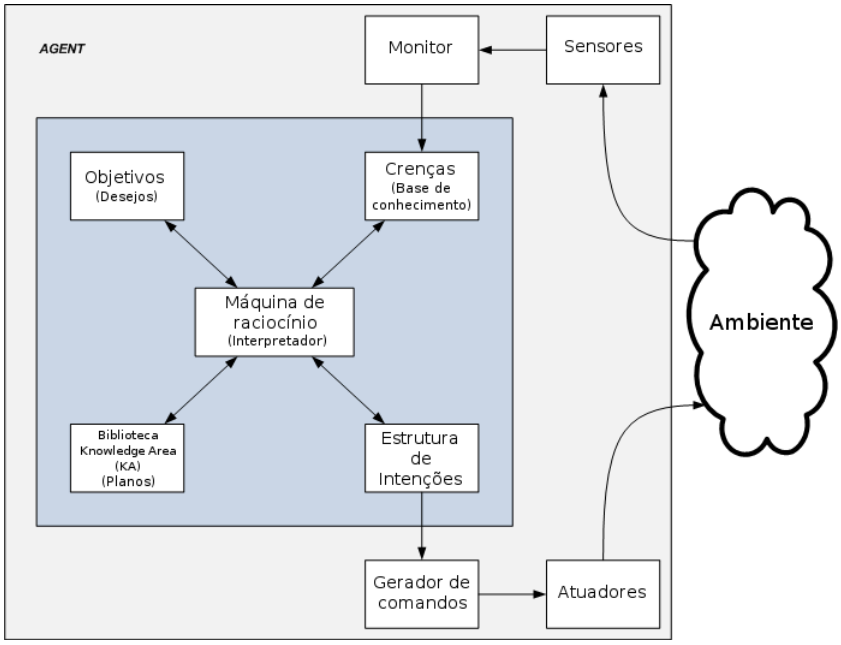

Figura 6: Organização do PRS. Fonte: Georgeff, M.P. e Ingrand, F.F.; 1989

Conforme vimos na figura 6, vários conceitos já conhecidos são aplicados. Podemos ver que é feita uma divisão clara das crenças, desejos e intenções do agente. Também vemos que a informação chega ao agente através de sensores, e sai dele por atuadores.

O interpretador é responsável por manter as crenças sobre o ambiente, escolher quais objetivos o agente deve tentar alcançar e qual área de conhecimento (*Knowledge Area ou KA*) que deve ser aplicada na situação atual.

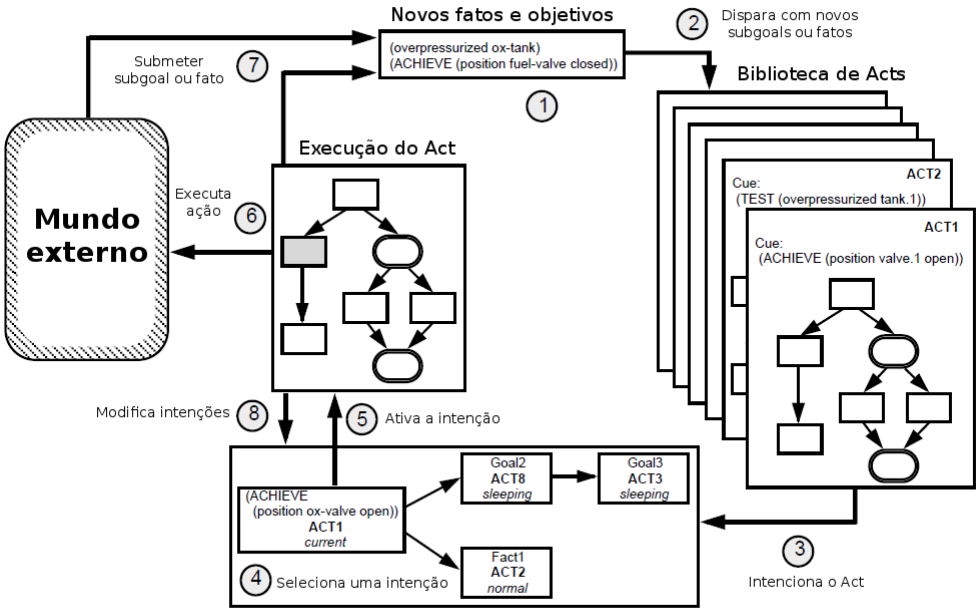

Figura 7: Ciclo de execução do Interpretador PRS. Fonte: Georgeff, M.P. e Ingrand, F.F.; 1989

De um ponto de vista conceitual, o interpretador opera de uma maneira relativamente simples. Dado um certo conjunto de objetivos e crenças no momento atual, existe um conjunto de áreas de conhecimento que são aplicáveis. O interpretador apenas escolhe uma delas para executar (GEORGEFF; LANSKY, 1987).

Uma das maiores vantagens do PRS é que, ao contrário da maioria das arquiteturas anteriores, ele permite que os planos sejam modificados ou descartados durante a atuação do agente no mundo, assim não é necessário criar o plano inteiro no início, e replanejar caso algo inesperado aconteça.

Durante a execução, novos objetivos podem surgir, e novas crenças são criadas. Quando isto acontece, o interpretador checa os bancos de dados para ver se existe uma nova *Knowledge Area* aplicável, e adapta o seu ciclo para esta nova situação.

3.2.2 JADE

O *Java Agent DEvelopment Framework* (JADE) é um *framework* implementado em *Java* que simplifica a implementação de sistemas multiagente *JADE*.

O *framework* é compatível com as especificações da *Foundation for Intelligent Physical*

Agents (FIPA)[4], uma organização sem fins lucrativos criada em 1996 que visa produzir padrões de software para agentes heterogêneos e interativos, e possui uma série de ferramentas para facilitar a produção e distribuição de sistemas multiagentes. A ferramenta permite que a plataforma sobre a qual os agentes estão rodando seja distribuída por várias máquinas, e estes podem transitar livremente de uma máquina para outra.

O sistema JADE pode ser descrito de duas maneiras. Por um lado, é uma ferramenta para rodar SMAs FIPA *compliant*, que permite que os agentes da aplicação façam uso das facilidades descritas pela especificação FIPA, como envio de mensagens e ciclo de vida de agentes (BELLIFEMINE; POGGI; RIMASSA, 2001). Por outro lado, o JADE pode ser usado como um framework *Java* que permite que o desenvolvedor acesse as utilidades FIPA através de abstrações orientadas à objetos.

O *framework* JADE foi criado para suportar qualquer tipo de agente, mas, uma outra ferramenta, o Jadex, voltada especificamente para agentes BDI foi construída baseada nele, com o objetivo de facilitar a criação de agentes desse tipo.

3.2.3 JADEX

Inicialmente baseado no JADE, a plataforma Jadex (BRAUBACH; POKAHR; LAMERSDORF, 2005) já evoluiu muito desde sua criação e se tornou uma ferramenta independente de igual ou maior importância que o próprio JADE.

Jadex é uma *engine* para sistemas multiagentes orientados ao modelo BDI que faz uso das linguagens XML e *Java* para a descrição dos agentes. Os agentes Jadex foram criados ao incorporar os conceitos do modelo BDI aos agentes JADE (BRAUBACH; POKAHR; LAMERSDORF, 2005). Esta ferramenta permite que você defina as crenças, desejos e intenções dos seus agentes em XML, e faça chamadas para métodos *Java*, que funcionam como os seus atuadores.

Além de descrever os agentes, o Jadex permite, através do sistema de regras *Jadex Rules*, que você execute os agentes de modo com que essas regras definam qual ação deve ser seguida. O *Jadex Rules* é uma engine de encadeamento de regras, e foi desenvolvido com o objetivo de ser um módulo de software reusável que pudesse ser facilmente integrado em outra aplicações. Ele possui uma versão que usa uma adaptação da linguagem *Java* para permitir que regras e condições sejam escritas de maneira intuitiva para quem está acostumado com a linguagem. Outro fator importante, é que o *Jadex Rules* possui um sistema de *debug* visual integrado, que é uma ferramenta indispensável na criação de grandes projetos.

[4]Página oficial: `http://www.fipa.org`.

Ambos JADE (BELLIFEMINE; POGGI; RIMASSA, 2001) e Jadex (BRAUBACH; POKAHR; LAMERSDORF, 2005) são plataformas implementadas em *Java*, e da mesma maneira que o Jason, podem ser executadas em *Applets Java*. Apesar disso, não encontramos nenhuma aplicação de sistema multiagente que rodasse totalmente em *browser*, porém, aplicações que executam na máquina do cliente, e apenas se comunicam com um servidor já são mais comuns.

3.3 CONCLUSÃO DA ANÁLISE

Como podemos ver através desta breve análise, a necessidade de se criar ferramentas e linguagens para o desenvolvimento de SMA não é algo novo, e muito trabalho já foi realizado sobre este problema. As ferramentas existentes são bastante consolidadas em sua área de aplicação, porém um nicho ainda não muito explorado é o de aplicações Web.

A proposta deste trabalho é utilizar este conhecimento já amadurecido, e com base nestas ferramentas e linguagens, criar uma nova ferramenta para desenvolvimento de SMA procurando atingir esse objetivo mais específico, visto que as grandes ferramentas existentes são limitadas à criação de agentes que rodam localmente.

4 AGENTES INTELIGENTES NA WEB

A Internet se tornou o meio mais conhecido para disseminação de informação e acesso a serviços nas redes de comunicação. Porém, devido ao rápido crescimento, alguns problemas surgiram, como por exemplo, localizar a informação desejada, otimizar o tráfego de dados e suportar trabalho coletivo (BOUDRIGA; OBAIDAT, 2004).

Os agentes inteligentes combinam tecnologias de computação e comunicação para oferecer soluções a estes problemas. Estes agentes são equipados com conceitos já citados neste trabalho, e utilizam técnicas para reagir a situações inesperadas e aprender através de experiência, melhorando a sua eficiência (ETZIONI; WELD, 1995).

Procurando satisfazer essa demanda em constante crescimento, podemos observar vários exemplos de aplicações de IA com agentes na Web, com alguns dos mais notáveis descritos a seguir.

4.1 APLICAÇÕES EXISTENTES NA WEB

4.1.1 SISTEMAS DE RECOMENDAÇÃO

Sistemas de recomendação são basicamente ferramentas que usam alguma técnica de filtragem de informação para encontrar dados que provavelmente são de interesse do usuário. A maior parte destes sistemas compara os perfis dos seus usuários, para tentar encontrar características que revelem informações sobre os seus interesses. Algumas características podem ser obtidas diretamente, e outras indiretamente.

As informações obtidas diretamente podem ser obtidas das seguintes maneiras, entre outras:

- Pedindo ao usuário para dar uma nota a um item.
- Pedindo para usuário ordenar uma lista de itens, do pior para o melhor.
- Pedindo para usuário criar uma lista de itens que ele gosta.

Informações obtidas indiretamente dependem de dados que não são informados diretamente pelo usuário, mas através de suas ações:

- Analisar os itens que usuário acessa em uma loja virtual.
- Analisar os itens que usuário compra em uma loja virtual.
- Analisar as redes sociais de um usuário e descobrir seus gostos.
- Analisar as redes sociais de um usuário e descobrir os gostos de pessoas que compartilham as suas comunidades.

Estes sistemas, muito usados em *sites* de *e-commerce* e redes sociais, são responsáveis por sugerir produtos e comunidades, e são altamente populares. Alguns dos maiores sites sociais e de informações, como, por exemplo, o *The Internet Movie Database* (IMDb), o *Last.FM*, *Rotten Tomatoes*, *Facebook*, *Ping*, entre vários outros, utilizam estes sistemas para melhorar a experiência dos seus usuários.

Um exemplo destes sistemas utilizando agentes inteligentes pode ser encontrado em (BIRUKOV; BLANZIERI; GIORGINI, 2005). A ideia dos autores é usar um agente para auxiliar o usuário na busca, onde este agente traça um perfil do usuário e a partir disto consegue distinguir quais itens do resultado de uma busca são de fato interessantes.

Mais do que isso, o sistema, chamado de *Implicit* é capaz de formar uma rede de agentes, fazendo uso de uma inteligência coletiva, denominada *Implicit Culture*. Essa rede parte do princípio que membros de uma mesma comunidade tendem a agir de maneira similar a outros membros, ou seja, o que é interessante para um, também será para os outros. Assim, quando você faz uma busca, o seu agente traz informações que outros usuários com interesses semelhantes aos seus acharam relevantes.

Agentes possuem uma grande aplicabilidade em sistemas de recomendação. É possível, por exemplo, ter agentes atribuídos a usuários e que aprendem sobre os mesmos e utilizam essas informações para obter melhores resultados. Uma mesma empresa que possui diversos sistemas de recomendação poderia também utilizar comunicação entre agentes de seus diferentes sistemas para obter informações de um mesmo usuário. Por exemplo, duas páginas diferentes, uma de venda de livros e outra de venda de filmes, poderiam usar comunicação entre agentes para obter informações de um cliente antigo de livros para lhe recomendar filmes que se encaixem em seu perfil na primeira vez que ele visita a loja virtual de filmes.

4.1.2 ENSINO A DISTÂNCIA

Algumas das aplicações mais interessantes na área de IAD tem como foco o ensino a distância, tentando usar agentes inteligentes para facilitar e customizar o aprendizado através

da internet.

(FENG, 2007) faz um estudo bastante detalhado sobre como sistemas multiagentes podem ajudar no ensino, e apresenta um modelo estrutural de aprendizado baseado na teoria do construtivismo. Este artigo cita que hoje a maioria dos cursos via Web seguem técnicas tradicionais de aprendizado, mas não conseguem recriar a interação entre alunos e professores. Sem esta interação, fica muito mais difícil alcançar um bom nível de aprendizado, além de dificultar ainda mais o processo de avaliação de um aluno.

Segundo os autores, agentes inteligentes apresentam uma solução melhor para este problema e é sugerido um sistema do tipo cliente-agente-servidor, onde o agente trabalha como uma espécie de mediador entre o conteúdo e o aluno.

Algumas vantagens desta ideia são apresentadas:

- Qualquer aluno pode estudar *online* quando quiser, e uma boa interface de comunicação aliada a uma base de dados pode inspirar os alunos.
- O aprendizado é adaptado especialmente para cada aluno, de acordo com a sua aptidão, através de análises obtidas pelos agentes, permitindo que diferentes estratégias de ensino sejam adotadas para cada aluno.
- Facilidade de manutenção e expansão, já que cada agente é uma entidade individual e é fácil de se aumentar o número destes agentes para atender um maior número de alunos.

4.2 DESENVOLVENDO APLICAÇÕES PARA A WEB

Desenvolver aplicações multiagentes para a Web não é muito diferente de desenvolvê-las para qualquer outra plataforma, já que a maioria das definições de agentes prega que estes devem ser o mais independentes possíveis, e serem capazes de transitar livremente de uma plataforma para outra.

4.2.1 FERRAMENTAS

Existem algumas ferramentas para auxiliar no desenvolvimento de aplicações específicas, como o *Seta2000*, descrito em (ARDISSONO, 2005). O *Seta2000* é uma infraestrutura para desenvolvimento de sistemas de recomendação, citados anteriormente.

Alguns exemplos de usos reais do *Seta2000* são apresentados no artigo, e fica claro que o sistema realmente funciona e é útil para auxiliar no desenvolvimento destes sistemas de

recomendação.

Internamente, o *Seta2000* faz uso de agentes BDI, mas como o seu foco é voltado para um uso específico, ele não permite que criemos agentes para uso em outros tipos de aplicações. Além disso, o sistema gerado é *server-side*, ou seja, exige que o criador tenha um servidor poderoso caso o número de usuários seja muito grande. Segundo os autores, uma das maiores fraquezas do trabalho é a quantidade de tempo gasta com comunicação cliente-servidor, o que pode atrasar bastante a aplicação final do ponto de vista do usuário.

Outra ferramenta existente é o JaCa-Web (MINOTTIE, 2010), uma implementação de um modelo para execução de agentes inteligentes em aplicações *clientside* na Web. O JaCa-Web é um *framework* que utiliza a linguagem de programação de agentes Jason, e o *framework* CArtAgO para programar o ambiente onde os agentes são executados.

Uma aplicação JaCa-Web é composta de três partes:

- Uma página Web HyperText Markup Language (HTML) com funções *JavaScript*;
- A *Applet* JaCa-Web, dividida em diferentes arquivos.
- Os componentes *JavaScript* do JaCa-Web, que são responsáveis pela camada de comunicação entre a *Applet* e o código *JavaScript* da página.

Tanto a plataforma proposta neste trabalho como o *framework* JaCa-Web têm como objetivo o desenvolvimento de aplicações Web, porém com algumas diferenças no modo como isto é implementado. O JaCa-Web utiliza *Applets*, que precisam ser aceitas pelo usuário e executam o código em uma máquina virtual, enquanto a plataforma proposta executa o código nativamente na máquina do cliente, dentro das restrições de segurança impostas pela tecnologia utilizada para permitir a execução (*Google NaCl*).

A plataforma proposta é apresentada com detalhes na sessão seguinte, com descrição das diversas etapas de implementação, metodologia e ferramentas utilizadas para seu desenvolvimento.

5 MODELO PROPOSTO

Nesta seção, detalharemos a proposta do trabalho, descrevendo a ideia principal, as ferramentas escolhidas para alcançar os objetivos e a metodologia utilizada para o desenvolvimento da plataforma.

Após o levantamento feito das linguagens e ferramentas existentes para desenvolvimento de SMAs, mais direcionado para ferramentas e aplicações já existentes para a Web, vimos que esse nicho é pouco explorado e possui um grande potencial. Fizemos então uma análise sobre as vantagens e desvantagens de algumas das linguagens disponíveis para o desenvolvimento da plataforma proposta.

5.1 FERRAMENTAS ESCOLHIDAS

Para iniciarmos o desenvolvimento da plataforma, alguns aspectos necessários para sua criação foram levantados:

- Linguagem utilizada pelos agentes;
- Linguagem para desenvolvimento da plataforma;
- Ambiente de desenvolvimento;
- Metodologia de desenvolvimento;
- Testes e validação da plataforma;

As escolhas feitas para satisfazer os aspectos acima serão descritas nas seções seguintes.

5.1.1 LINGUAGEM DE DESCRIÇÃO DOS AGENTES

A linguagem escolhida para ser utilizada como base da plataforma foi a 3APL-M. O principal fator que nos levou a essa escolha foi o fato de que a linguagem é voltada para dispositivos móveis, que possuem uma capacidade menor, e por isso é uma linguagem simplificada e leve, ideal para o desenvolvimento de agentes para a Web.

Outro fator importante considerado foi que a 3APL-M possui o código-fonte de interpretadores escritos em *Java* e em *Haskell* disponíveis na página da linguagem, o que nos ajudou a criar nossa própria plataforma a partir da versão em *Java* encontrada.

5.1.2 LINGUAGEM DA PLATAFORMA

Para desenvolver a plataforma, foram estudadas duas abordagens diferentes. A primeira delas é utilizar a linguagem *JavaScript*, que é a mais comum para uso em *browsers* e realiza o processamento no lado do cliente. Boa parte de sua utilização é na criação de páginas dinâmicas e interfaces mais atraentes aos usuários, mas também pode ser utilizada na criação das mais diversas aplicações. A segunda opção é desenvolver a plataforma na linguagem C++ e utilizar o *Google Native Client* (NaCl), que é uma tecnologia *open-source* com o objetivo de permitir a execução de código nativo por aplicações Web. Outra opção existente era a utilização da linguagem *Java* em conjunto com alguma de suas tecnologias, como *Applets*, por exemplo, mas um dos objetivos do trabalho era fornecer uma alternativa a *Java*, que está presente em diversas plataformas de SMA.

Considerando a opção da utilização de *JavaScript*, a grande estabilidade da linguagem seria um dos pontos fortes de sua escolha, além de ser uma linguagem de *script*[1] amplamente utilizada e documentada, facilitando a integração da plataforma com as aplicações sendo desenvolvidas, já que seriam todas desenvolvidas em *JavaScript*. Por outro lado, alguns dos problemas encontrados nessa abordagem são a dificuldade de realizar testes em programas mais complexos desenvolvidos em *JavaScript* e a ausência de suporte a *threads* na linguagem.

A utilização da tecnologia *Google NaCl* tem como principal vantagem permitir o desenvolvimento da plataforma na linguagem C++, que é também muito conhecida e utilizada nos mais diversos tipos de aplicações, além de possuir suporte a *threads* através de bibliotecas já consolidadas. Por outro lado, a tecnologia *Google NaCl* ainda é recente e não é tão utilizada, apesar de já existirem diversas aplicações de exemplo disponíveis.

Dentre as vantagens e desvantagens levantadas entre as duas abordagens, a opção de utilizar o *Google NaCl* se enquadrou melhor em nossas expectativas no desenvolvimento da plataforma. Mais detalhes do funcionamento e características da tecnologia serão descritas a seguir.

[1] Linguagens de *Script* são linguagens interpretadas que são executadas dentro de programas ou outras linguagens de programação, estendendo as funcionalidades de tal programa.

5.1.3 GOOGLE NATIVE CLIENT

O *Google Native Client* é uma plataforma de código aberto de *sandboxing* para execução segura de código nativo não confiável através de um *browser*. Essa tecnologia permite que aplicações Web com uma grande demanda de processamento ou de interatividade, como jogos, aplicações multimídia e análise e visualização de dados, sejam executadas na máquina do cliente com acesso restrito, formando um ambiente seguro (GOOGLE-CODE, Acessado em Maio de 2011). Atualmente a tecnologia *Google NaCL* só funciona no browser *Google Chrome*.

O projeto NaCl possui um *kit* de desenvolvimento, o *Native Client Software Development Kit (SDK)* que permite criar executáveis a partir de aplicações já existentes ou do zero, e contém um grupo de ferramentas com versões customizadas do compilador das linguagens C e C++, a ferramenta para *debug GNU Debugger* (GDB), algumas bibliotecas, vários exemplos e tutoriais. O objetivo do projeto é permitir o uso de outras linguagens de programação, mas atualmente apenas as linguagens C e C++ são suportadas.

Uma típica aplicação que utiliza o *Google NaCL*, chamada de módulo, possui ao menos três entidades básicas:

- Uma página HTML, que inclui código *JavaScript* e é a interface com o restante do módulo;
- Um arquivo *Native Client executable* (.nexe), que é o módulo NaCl em C ou C++, as linguagens suportadas até o momento pela tecnologia. Esse módulo também pode utilizar a biblioteca *Pepper*, que é uma coleção de interfaces e funções inclusa no SDK que fazem as conversões necessárias na comunicação entre *JavaScript* e o módulo NaCl.
- Um arquivo do tipo *manifest* usado pelo sistema para determinar qual módulo NaCl compilado deve ser carregado para uma determinada plataforma alvo (32 ou 64 bits).

A página HTML é responsável por carregar o módulo NaCl através de um pequeno trecho incluso no arquivo, como no exemplo a seguir:

```
<embed name="nacl_module"
         id="hello_world"
         width=0 height=0
         nacl="hello_world.nmf"
         type="application/x-nacl"
         onload="moduleDidLoad();" />
```

Essa *tag* na página HTML irá carregar o módulo *hello_world* e executar a função *JavaScript moduleDidLoad()*, que apenas exibe a mensagem "*SUCCESS*" na página caso o módulo tenha sido carregado corretamente.

Independentemente da linguagem de programação utilizada, o fluxo de execução de um módulo segue as etapas observadas na figura 8 para garantir que o código do módulo não oferece ameaças à segurança da máquina acessando a aplicação. A partir do módulo compilado, o arquivo binário é validado e o módulo não é executado caso seja encontrada alguma irregularidade. Durante a execução do módulo, caso exista alguma tentativa de uma atividade suspeita, o módulo é interrompido. Caso nenhum problema seja detectado nessas duas etapas, o módulo é terminado com segurança após sua execução.

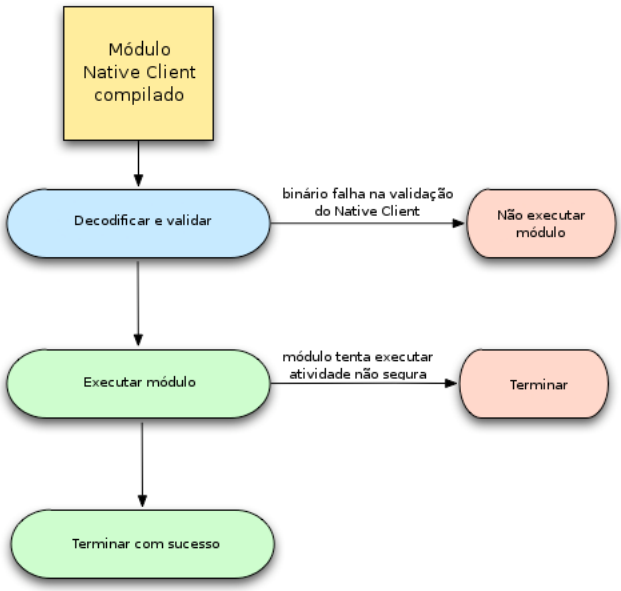

Figura 8: Sistema de *runtime* do NaCl. Fonte: página do *Google Native Client*

Algumas das atividades consideradas suspeitas pela tecnologia NaCl são a manipulação de dispositivos ou de arquivos diretamente (existe uma *Application Programming Interface* (API) especial para lidar com arquivos), acesso direto ao sistema operacional ou uso de código auto modificável para esconder outras intenções do código, como escrever em regiões protegidas de memória.

O NaCl inclui um subsistema de *runtime* que provê uma interface reduzida de chamadas de sistema e abstrações de recursos para isolar os códigos executáveis da máquina hospedeira. Os arquivos executáveis gerados (os arquivos .nexe) são carregados através do *sel_ldr* (*secure ELF loader*) e são lançados como um processo separado, que então se comunica com o *plugin* NaCl.

Uma visão mais abstrata da estrutura de um módulo NaCl pode ser vista na figura 9.

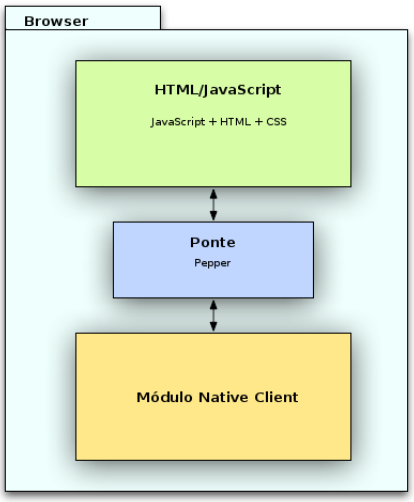

Figura 9: Estrutura de um módulo NaCl. Fonte: página do *Google Native Client*

Um exemplo típico de funcionamento de um módulo pode ser descrito numa série de passos, de acordo com a figura 9:

- Na página HTML, são criados um campo de texto, que recebe uma entrada do usuário, e um botão que dispara o módulo.
- Ao clicar nesse botão, uma função *JavaScript* na página HTML chama um método criado na ponte entre *browser* e módulo. Essa ponte é um arquivo na linguagem C e gerado automaticamente ao utilizar ferramentas do *Google NaCL* SDK. Além das funções já existentes na geração automática desse arquivo, é necessário criar identificadores para os métodos que serão chamados no módulo, e é chamada a função correspondente. O parâmetro inserido pelo usuário no campo de texto utiliza a API Pepper para converter esse parâmetro para que se torne apropriado para o módulo.
- A função do módulo é chamada e executada, sendo interrompida caso tente executar alguma atividade suspeita, conforme visto anteriormente.
- O resultado da execução do módulo também é convertido através da API Pepper para retornar ao *browser*.

Essa tecnologia se preocupa em manter a neutralidade de *browsers*, o que também conta positivamente para atingirmos um dos objetivos do nosso trabalho, que é permitir uma expansão do uso de agentes em aplicações, e apesar de ser um projeto recente, já pode ser utilizado em

máquinas x86 de 32 e 64 bits com os sistemas operacionais Windows, Mac OS X e Linux, o que amplia ainda mais o público-alvo da nossa plataforma.

5.1.4 AMBIENTE DE DESENVOLVIMENTO

Durante o desenvolvimento do projeto foi utilizada a *Integrated Development Environment* (IDE) Eclipse para a codificação em C++ da *engine* 3APL-M, a *engine* mProlog o código de integração do NaCl e na criação dos testes da aplicação.

Para a geração da plataforma executável, foi utilizado o compilador nacl-g++, disponibilizado no SDK do *Google NaCL*.

Devido ao caráter experimental da ferramenta NaCl, os testes das aplicações de exemplo só foram executados no browser *Google Chrome* versão 12.

5.2 ESTRUTURA

A estrutura básica da plataforma proposta pode ser vista como duas camadas separadas, conforme figura 10.

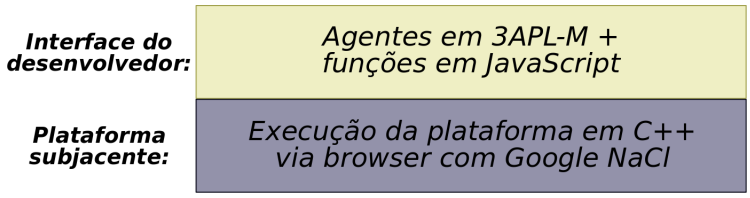

Figura 10: Estrutura básica da plataforma.

Na camada de interface do desenvolvedor, usuário da plataforma, o conhecimento necessário para criar os agentes é da linguagem de descrição de agentes 3APL-M e de funções *JavaScript* que serão utilizadas pelos agentes para realizar seus objetivos. A plataforma em si, representada na figura como plataforma subjacente, fica transparente ao desenvolvedor, e é responsável pela execução dos agentes.

Como já mencionado, a plataforma é escrita na linguagem de programação C++ e com a ajuda da tecnologia *Google NaCL* é executada na máquina do cliente via browser. As funções escritas em *JavaScript* também usam *bridges* através do *Google NaCL* para serem executadas durante o processamento de um agente. O ciclo de funcionamento da plataforma como um todo pode ser visto na figura 11.

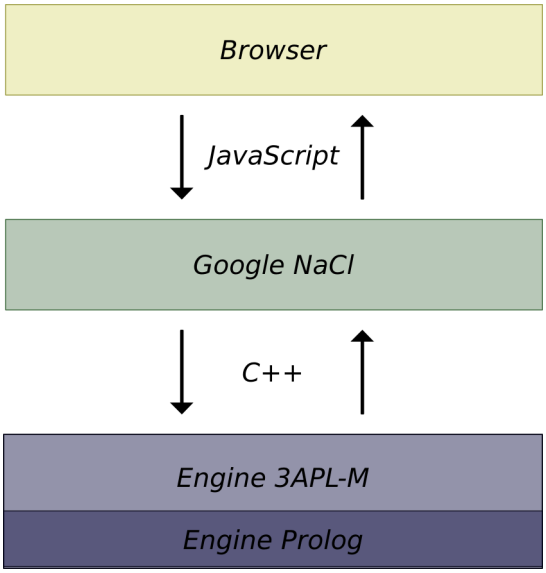

Figura 11: Ciclo de funcionamento da plataforma.

O *browser* do cliente se comunica através de funções *JavaScript* com a tecnologia *Google NaCL*, que executa os agentes na máquina do cliente na plataforma de agentes escrita na linguagem C++, que é composta da plataforma de execução de agentes descritos em 3APL-M e uma *engine* Prolog, responsável pelos passos de deliberação de um agente. Os resultados obtidos na deliberação são enviados novamente ao *browser* através de funções *JavaScript*, com o auxílio da tecnologia *Google NaCL*.

A arquitetura da *engine* 3APL-M escrita em C++ se manteve basicamente a mesma que a da plataforma original, em *Java*, como pode ser visto na figura 12.

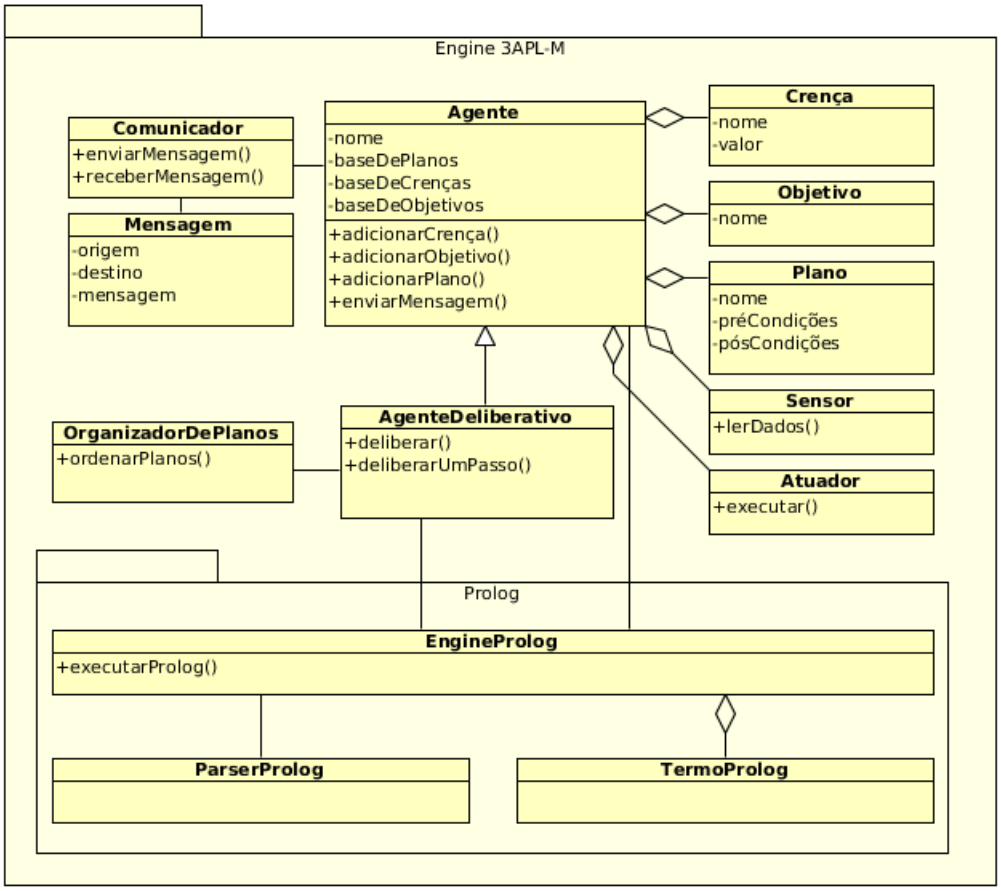

Figura 12: Arquitetura simplificada da *engine* 3APL-M.

Como pode ser visto na figura 12, um objeto da classe Agente possui um nome e uma base de planos, uma base de crenças e uma base de objetivos, e alguns métodos que modificam essas estruturas, que são definidas inicialmente no arquivo de descrição do agente em 3APL-M. Cada uma dessas estruturas de um agente é uma coleção de objetos de seu respectivo tipo, conforme ilustra a figura. Por exemplo, a *baseDePlanos* de um agente é composta por uma coleção de objetos do tipo *Plano*, que possuem um nome, uma pré-condição e uma pós-condição, definidos na linguagem 3APL-M.

Além dessas estruturas básicas responsáveis pelos estados mentais de um agente, é possível utilizar três mecanismos importantes para a execução de diversas tarefas pelos agentes, os Sensores, Atuadores e Comunicadores. Os dois primeiros mecanismos são destinados à interação de um agente com o ambiente, percebendo o estado atual e recebendo estímulos do ambiente através de Sensores e causando modificações nesse mesmo ambiente através de Atuadores. O mecanismo Comunicador é responsável pela troca de mensagens entre agentes dentro de um mesmo sistema, e a implementação atual da linguagem 3APL-M permite alguns tipos de mensagens entre agentes:

- Mensagens do tipo *INFORM*: Permite que um agente A envie a um agente B alguma informação nova, adicionando alguma crença à base de crenças do agente B;
- Mensagens do tipo *REQUEST*: Permite que um agente A envie um pedido a um agente B, adicionando um objetivo à base de objetivos do agente B.

Com relação à arquitetura como um todo, a mudança mais notável entre a implementação original e a plataforma implementada veio através de uma camada extra de abstração, que tem como objetivo representar toda a comunicação do mundo exterior com a *engine*. Esta camada foi criada utilizando o padrão de *design* "Fachada"(GAMMA, 1994). A ideia é que toda a comunicação feita entre o módulo NaCl e a *engine* 3APL-M seja feita através desta classe, assim garantimos que o ambiente exterior tenha conhecimento somente da interface exportada nesta fachada, sem se importar em como estas funções são implementadas internamente.

Esta integração pode ser vista no diagrama na figura 13:

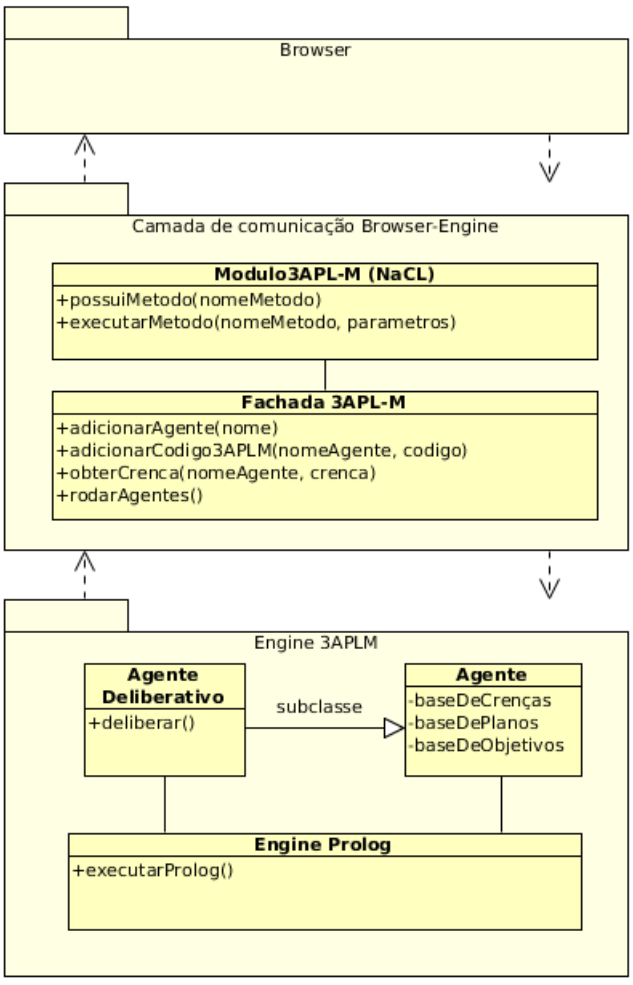

Figura 13: Integração entre componentes da plataforma.

Durante o processo de criação de uma aplicação com SMA, o desenvolvedor que utiliza a plataforma proposta deve incluir no código de sua página HTML funções *JavaScript* que chamam as funções definidas na interface citada, exemplificada na figura 13 com as funções *adicionarAgente*, *adicionarCodigo3APLM*, *obterCrenca* e *rodarAgentes*. A camada intermediária da arquitetura que deve se responsabilizar por verificar se as funções chamadas existem e estão com os parâmetros adequados, além de realizar a chamada efetiva para a *engine* em C++ subjacente. O processo de retorno da execução da aplicação também passa pela camada intermediária para retornar ao *browser* e ser exibido ao usuário final.

Após definida a estrutura e arquitetura básica da plataforma, separamos o desenvolvimento da mesma em algumas etapas, descritas na seção seguinte.

5.3 METODOLOGIA DE DESENVOLVIMENTO

Após definidas as linguagens e o ambiente de desenvolvimento a serem utilizados no projeto, separamos a criação da plataforma em algumas etapas:

1. Análise da linguagem de descrição de agentes 3APL-M a fim de remover, adicionar ou modificar aspectos que serão mais adequadas à nossa plataforma;
2. Estudo de características do desenvolvimento com uso da tecnologia *Google NaCl*, incluindo familiarização com as ferramentas disponíveis e aplicação de tutoriais;
3. Utilização do código-fonte do interpretador desenvolvido em *Java* da linguagem de agentes 3APL-M, disponível na página do projeto, como base para nossa própria plataforma, adaptando os procedimentos que envolvem o motor de inferência e outras características para nossos objetivos;
4. Finalização da codificação da plataforma na linguagem C++ em conjunto com a tecnologia *Google NaCl*;
5. Criação de uma aplicação para testes da plataforma desenvolvida;

Para as etapas de análise da linguagem de descrição de agentes 3APL-M e de seu interpretador em *Java* disponível, desenvolvemos uma aplicação razoavelmente complexa nesse ambiente (versão para *Java* SE) juntamente com um estudo do código do interpretador e da linguagem em si, procurando entender na prática alguns dos conceitos empregados e verificar quais aspectos são ou não relevantes para nossa plataforma. Ao seguir com essa prática, um dos aspectos que optamos por adicionar foi o cálculo de utilidade de planos.

A utilidade de um plano é um valor numérico obtido a partir do custo de uma capacidade do agente (*capability*) e do valor atribuído ao seu objetivo (*goal*). A linguagem 3APL-M propôs o cálculo de utilidade fazendo uma subtração entre valores fixos para cada capacidade e objetivo. Esta abordagem mais simples foi implementada para nossa plataforma, porém, realizar este cálculo através de funções permitiria uma representação mais realista dessa utilidade, já que ela pode ser diferente dependendo do estado atual do agente, que é definido por suas crenças atuais (*beliefs*). A utilização de funções para este cálculo é uma das funcionalidades planejadas para nossa plataforma.

5.3.1 CONFIGURAÇÃO DO AMBIENTE

Nesta primeira etapa foi pesquisada a necessidade de ferramentas para as demais etapas do desenvolvimento da plataforma, como codificação em C++, integração com o *Google NaCL*, utilização de ferramentas específicas, testes em *browsers* e demais etapas do projeto. Algumas pequenas restrições foram encontradas, como:

- Instalação do *Google NaCl SDK*, atualmente na versão 0.3;
- Utilização do *browser Google Chrome* na versão 12 para permitir a execução da plataforma;

O ambiente de desenvolvimento utilizado não apresentou nenhum tipo de restrição nem necessitou de ferramenta específica para codificação da plataforma.

5.3.2 CODIFICAÇÃO DA PLATAFORMA EM C++

A etapa de implementação da plataforma em C++, composto da plataforma de execução de agentes 3APL-M e o motor *Prolog* incluso para o processo deliberativo dos agentes, foi a mais longa entre as etapas. Partindo da implementação de uma plataforma escrita em *Java*, disponibilizada na própria página do projeto 3APL-M, foi necessário adaptar diversas chamadas de métodos, estruturas de dados e aspectos diferentes entre as linguagens *Java* e C++.

Para implementar as estruturas de dados presentes na plataforma, foram utilizadas as bibliotecas que fazem parte da biblioteca padrão da linguagem C++, como por exemplo $<string>$, $<map>$ e $<vector>$. As classes, funções e métodos implementados foram testados unitariamente e em conjunto para assegurar que a execução dos agentes teria um resultado válido. Como resultado dessa etapa do projeto, foi obtida uma plataforma para execução de agentes 3APL-M *standalone*, pronta para ser utilizada para criar aplicações de SMA em C++.

Para permitir a execução de mútiplos agentes simultaneamente, foi utilizada a biblioteca *<pthread>*. Essa biblioteca provê um conjunto de funções para suportar aplicações com múltiplos fluxos de execução, característica presente em sistemas multiagentes.

MUDANÇAS NA PLATAFORMA

Algumas funcionalidades foram adicionadas, removidas e modificadas com relação à plataforma em *Java* utilizada como base no que se refere à linguagem de descrição de agentes. Na plataforma *Java*, algumas funções são incluídas aos agentes como funções nativas ou *built-in* que podiam ser usadas para obter a quantidade de memória sendo utilizada na execução, entre outras coisas. A maioria dessas funções não era necessária para a plataforma sendo desenvolvida em C++ e foram retiradas.

A plataforma *Java* também possuía várias chamadas do coletor de lixo da linguagem, para melhorar a questão de performance. A linguagem C++ não possui esse sistema de coletor de lixo nativamente, e, portanto, essa funcionalidade também foi retirada de nossa plataforma.

Por outro lado, foram adicionadas funções que permitiam a classificação de planos de um agente conforme sua utilidade. Para isso, foi concluída a etapa de análise da descrição do agente para as estruturas *WORTHBASE* e *COSTBASE*, que não estavam finalizadas na plataforma *Java*. A primeira estrutura permite a atribuição de valores para os objetivos (*goals*) de um agente, e a segunda permite atribuir os custos de suas capacidades (*capabilities*). Com isso, ao encontrar mais de um plano possível para chegar a um determinado objetivo, o agente consegue calcular, de acordo com o custo de cada capacidade envolvida nos planos possíveis, qual dos planos será o com maior utilidade (diferença entre valor do objetivo e custo do plano).

5.3.3 INTEGRAÇÃO COM O NACL

Para poder utilizar o *Google NaCL*, é preciso habilitar essa tecnologia no navegador *Google Chrome*, através de uma opção do próprio navegador. Além disso, o *Google NaCL* SDK fornece também um script na linguagem *Python* para executar um pequeno *web server*, necessário para execução dos módulos desenvolvidos com a tecnologia, possibilitando testes locais. Após a habilitação do navegador e a inicialização do *web server*, já é possível testar o módulo em desenvolvimento.

Com a finalização da plataforma de agentes na linguagem C++, a etapa de integração com o NaCl pode ser iniciada. O código foi recompilado com o compilador disponibilizado no *Google NaCL* SDK, que gera também um arquivo C++ que serve como ponte entre o *browser* e o código

C++ do módulo desenvolvido (a plataforma de agentes). Esse arquivo adicional gerado deve ser modificado para permitir a chamada de funções do módulo criado. No nosso caso, foi preciso adicionar um método responsável por criar e executar um agente, passando a sua descrição em 3APL-M como parâmetro.

Com o *web server* em execução, basta carregar no *browser* uma página HTML que se comunique com o módulo desenvolvido. O *Google NaCL* SDK também gera automaticamente um arquivo de página HTML para ser utilizado para esse fim, basta modificar o arquivo para adicionar a estrutura de sua página HTML como qualquer outra (textos, botões, campos, imagens, etc.) e adicionar funções *JavaScript* que chamem os métodos criados no arquivo que liga *browser* e módulo.

Foram realizados alguns testes para assegurar que a plataforma continuava apresentando o mesmo funcionamento observado na versão anterior à integração com o *Google NaCL*, implementada puramente em C++. Algumas pequenas modificações foram necessárias para permitir que os resultados gerados durante a execução dos agentes fosse visível no *browser*. Por exemplo, a plataforma em C++ utilizava a saída padrão (*output stream*) para imprimir os passos de deliberação dos agentes. Isso foi modificado para que esses resultados fossem impressos no *browser* em um local específico na página do módulo.

5.3.4 DESENVOLVIMENTO DE EXEMPLOS

Apesar do foco deste trabalho não ser a implementação de aplicações multiagente, mas sim de uma plataforma de desenvolvimento, esta plataforma tem como objetivo permitir uma transição fácil de desenvolvedores Web para desenvolvedores de sistemas multiagentes. Para facilitar nesta transição, foram desenvolvidas algumas aplicações de exemplo, que podem ser seguidas como uma espécie de modelo para a aplicação que o usuário deseja criar.

Os exemplos visam contemplar alguns tipos comuns de aplicações, e variam de exemplos bem simples, com agentes do tipo *Hello World*, até alguns mais complexos, que exemplificam melhor as possibilidades dos agentes descritos pela linguagem 3APL-M. Usos mais realistas da plataforma envolvem uso extensivo de chamadas de funções *JavaScript* definidas pelo desenvolvedor, e, dependendo da aplicação sendo criada, a comunicação entre os agentes.

Foi criada uma página de exemplo que recebe a descrição 3APL-M de um agente e o executa. A página HTML, conforme a figura 14, contém uma área de texto para que seja inserida a descrição, um botão que dispara o agente e uma região onde os passos da deliberação dos agentes são escritos.

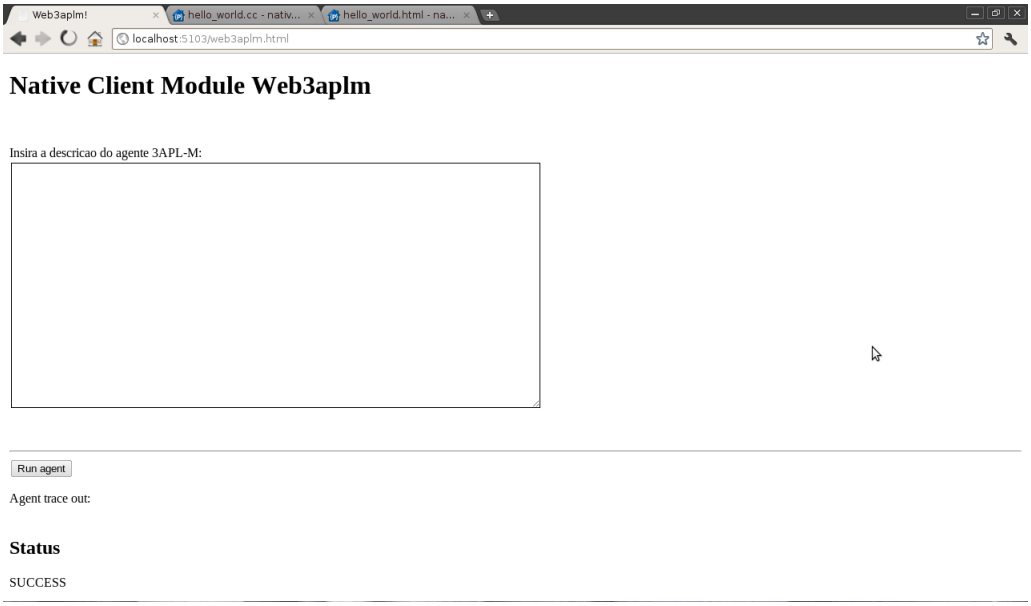

Figura 14: Página HTML para testes da plataforma.

Para exemplificar, foi criado um agente simples em 3APL-M para ser utilizado na página HTML criada. Na figura 15 é possível ver a descrição em 3APL-M e uma parte do resultado do processo de deliberação.

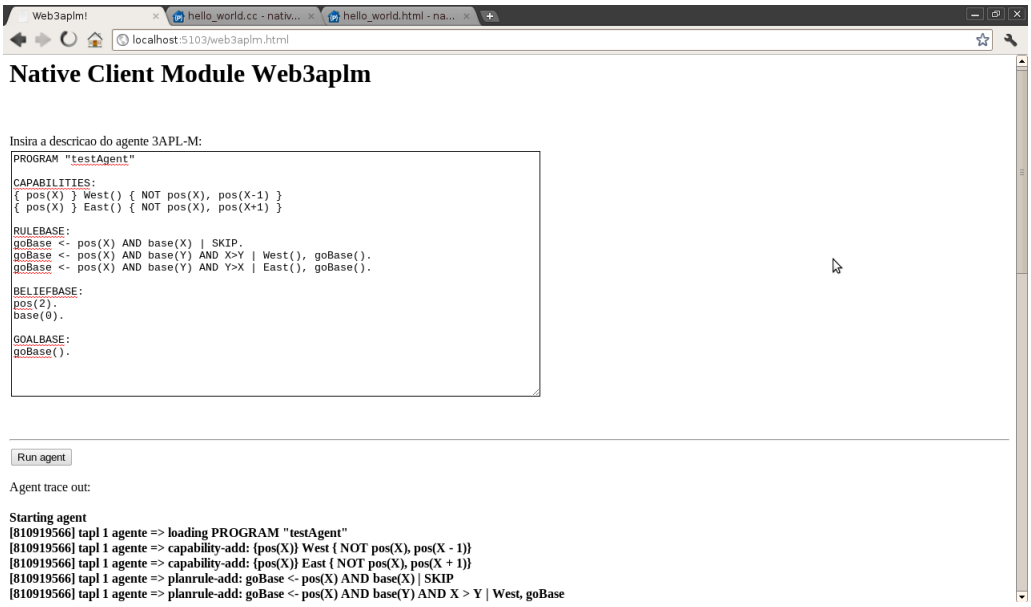

Figura 15: Exemplo de um agente simples.

A saída completa do processo de deliberação pode ser vista abaixo. Esse agente simples tinha apenas duas capacidades, andar para a direita (*East()*) e andar para a esquerda (*West()*).

Sua base de crenças inicial continha a informação de sua posição como sendo ”2” (*pos(2)*) e a posição da base como sendo ”0” (*base(0)*). Seu objetivo era chegar até a base (*goBase()*) e isso pode ser atingido através de dois planos, que levam em consideração a direção em que a base se encontra.

```
[810919566] tapl 1 agente => loading PROGRAM "testAgent"
[810919566] tapl 1 agente => capability-add: {pos(X)} West { NOT pos(X), pos(X - 1)}
[810919566] tapl 1 agente => capability-add: {pos(X)} East { NOT pos(X), pos(X + 1)}
[810919566] tapl 1 agente => planrule-add: goBase <- pos(X) AND base(X) | SKIP
[810919566] tapl 1 agente => planrule-add: goBase <- pos(X) AND base(Y) AND X > Y | West, goBase
[810919566] tapl 1 agente => planrule-add: goBase <- pos(X) AND base(Y) AND Y > X | East, goBase
[810919566] tapl 1 agente => belief-add: pos(2)
[810919566] tapl 1 agente => belief-add: base(0)
[810919566] tapl 1 agente => goal-add: goBase
[810919566] tapl 1 agente => deliberating: [goBase]
[810919567] tapl 1 agente => === deliberation-step ===
[810919567] tapl 3 agente => goal-base contains: goBase
[810919567] tapl 2 agente => planrule-find-selected: (1) goBase <- pos(X) AND base(X) | SKIP
[810919567] tapl 2 agente => planrule-find-selected: (2) goBase <- pos(X) AND base(Y) AND X > Y | West, goBase
[810919567] tapl 2 agente => planrule-find-selected: (3) goBase <- pos(X) AND base(Y) AND Y > X | East, goBase
[810919567] tapl 1 agente => planbase-add: goBase <- TRUE | West, goBase
[810919567] tapl 1 agente => plan-classified: (1)= goBase <- TRUE | West, goBase: utility=0
[810919567] tapl 2 agente => planbase-result: goBase <- TRUE | West, goBase
[810919567] tapl 3 agente => executing plan: goBase <- TRUE | West, goBase
[810919568] tapl 1 agente => goal-add-in-front: goBase
[810919568] tapl 1 agente => goal-add-in-front: West
[810919568] tapl 1 agente => === deliberation-step ===
[810919568] tapl 3 agente => goal-base contains: West
[810919568] tapl 3 agente => goal-base contains: goBase
[810919568] tapl 3 agente => capability-execute: {pos(X)} West { NOT pos(X), pos(X - 1)} with X=2
[810919568] tapl 1 agente => belief-delete: pos(2)
[810919569] tapl 1 agente => belief-add: pos(1)
[810919569] tapl 1 agente => === deliberation-step ===
[810919569] tapl 3 agente => goal-base contains: goBase
[810919569] tapl 2 agente => planrule-find-selected: (1) goBase <- pos(X) AND base(X) | SKIP
[810919569] tapl 2 agente => planrule-find-selected: (2) goBase <- pos(X) AND base(Y) AND X > Y | West, goBase
[810919569] tapl 2 agente => planrule-find-selected: (3) goBase <- pos(X) AND base(Y) AND Y > X | East, goBase
[810919569] tapl 1 agente => planbase-add: goBase <- TRUE | West, goBase
[810919569] tapl 1 agente => plan-classified: (1)= goBase <- TRUE | West, goBase: utility=0
[810919570] tapl 2 agente => planbase-result: goBase <- TRUE | West, goBase
[810919570] tapl 3 agente => executing plan: goBase <- TRUE | West, goBase
[810919570] tapl 1 agente => goal-add-in-front: goBase
[810919570] tapl 1 agente => goal-add-in-front: West
[810919570] tapl 1 agente => === deliberation-step ===
[810919571] tapl 3 agente => goal-base contains: West
[810919571] tapl 3 agente => goal-base contains: goBase
[810919571] tapl 3 agente => capability-execute: {pos(X)} West { NOT pos(X), pos(X - 1)} with X=1
[810919571] tapl 1 agente => belief-delete: pos(1)
[810919571] tapl 1 agente => belief-add: pos(0)
[810919572] tapl 1 agente => === deliberation-step ===
[810919572] tapl 3 agente => goal-base contains: goBase
[810919572] tapl 2 agente => planrule-find-selected: (1) goBase <- pos(X) AND base(X) | SKIP
[810919572] tapl 2 agente => planrule-find-selected: (2) goBase <- pos(X) AND base(Y) AND X > Y | West, goBase
```

```
[810919572] tapl 2 agente => planrule-find-selected: (3) goBase <- pos(X) AND base(Y) AND Y > X | East, goBase
[810919572] tapl 1 agente => planbase-add: goBase <- TRUE | SKIP
[810919572] tapl 1 agente => plan-classified: (1)= goBase <- TRUE | SKIP: utility=0
[810919573] tapl 2 agente => planbase-result: goBase <- TRUE | SKIP
[810919573] tapl 3 agente => executing plan: goBase <- TRUE | SKIP
[810919573] tapl 1 agente => goal-add-in-front: SKIP
[810919573] tapl 1 agente => === deliberation-step ===
[810919574] tapl 3 agente => goal-base contains: SKIP
```

5.3.5 TESTES E VALIDAÇÃO

Durante o projeto foram realizados vários testes para ajudar no desenvolvimento da plataforma, sejam estes referentes ao código em C++, quanto a integração com o NaCl e finalmente com testes de aceitação envolvendo o modelo completo, contando com uma página real sendo executada em um *browser*.

No começo, estes testes basicamente eram feitos comparando os resultados obtidos da plataforma C++ com os resultados esperados, da plataforma original em *Java*. Devido à natureza da plataforma original, não foram necessários muitos testes para cobrir uma grande parte do código, já que a plataforma em si é simples e possui várias dependências internas, permitindo a execução de testes em pontos chave que cobrem uma variedade de aspectos da mesma.

Após o término da plataforma em C++, foram realizados testes de aceitação da plataforma em si, sem nenhuma conexão com NaCl ou *browsers*. Estes testes tinham como objetivo testar somente a execução dos agentes dentro da plataforma, esperando um funcionamento igual ao da plataforma em *Java* utilizada como base.

Durante a integração da plataforma com o *Google NaCL*, inicialmente foram feitos alguns testes sobre a tecnologia em si, seguindo alguns exemplos disponíveis na página do projeto, para depois ser testada a integração dela com nossa plataforma. Testes de integração deste tipo consistiam basicamente em verificar que era possível fazer a comunicação entre a plataforma e o *browser* através do NaCl.

Por último, foram feitos testes com o sistema inteiro em funcionamento, envolvendo todas as etapas já testadas até então simulando um caso de uso real da plataforma, sendo utilizada para desenvolver um aplicação de agentes.

5.4 CRIANDO UMA APLICAÇÃO COM A PLATAFORMA

Nesta seção será apresentada uma maneira de se implementar um sistema multiagente na Web utilizando a plataforma proposta.

Como foi citado anteriormente, o usuário final da plataforma não precisa saber como a plataforma é implementada, e portanto, não precisa conhecer a linguagem C++. Tudo o que é visível para ele é a interface provida pela plataforma.

Assumimos que o desenvolvedor já possui conhecimento da linguagem *JavaScript*, assim como prática no desenvolvimento de aplicações Web, portanto, esta seção vai ser focada apenas nas tarefas necessárias para se integrar uma aplicação Web com um SMA rodando na plataforma. Estas tarefas se resumem basicamente a incluir o código dos agentes e chamadas de métodos da plataforma dentro do código *JavaScript*, e manter o arquivo *.nexe* (distribuição da plataforma compilada) junto com a aplicação.

A tecnologia *Google NaCL* exige pouca intervenção no código *JavaScript* do desenvolvedor. Para permitir chamadas à plataforma, basta criar uma *tag "embed"* dentro do arquivo HTML principal da aplicação, que será responsável por carregar o código pré-compilado da plataforma. Após a plataforma ser carregada, ela pode ser acessada diretamente como um objeto *JavaScript*.

Um exemplo do código inserido no arquivo HTML pode ser visto a seguir:

```
<html>
...

Plataforma3APLM = null;

function onLoad() {
    Plataforma3APLM = document.getElementById('web3aplm');
}

...

<embed
    name="nacl_module"
    id="web3aplm"
    width=0 height=0
    nacl="web3aplm.nmf"
    type="application/x-nacl"
    onload="onLoad();"
/>
```

```
...
</html>
```

Neste exemplo definimos uma váriavel global ”Plataforma3APLM”que representa a plataforma. Esta váriavel é inicializada na função *”onLoad”*, que é chamada assim que o *NaCl* termina de carregar o módulo descrito pelo arquivo de manifesto *”web3aplm.nmf”*.

Com o módulo carregado, o desenvolvedor tem acesso às funções exportadas na interface da plataforma, como pode ser visto a seguir:

```
function lançarAgentes() {
    Plataforma3APLM.lançarAgentes()
}

function criarAgente() {
    codigo_do_agente = "..."
    Plataforma3APLM.criarAgente("nome")
    Plataforma3APLM.carregarCodigo3APLM("nome", codigo_do_agente)
}
```

Estas funções podem ser definidas em qualquer lugar, e podem ser executadas a qualquer momento, como qualquer outra função *JavaScript*. Vale notar que a função *criarAgente* recebe um parâmetro *string* com o código do agente para a plataforma. Esta é uma *string JavaScript* normal, e pode estar descrita diretamente dentro da função, em outro arquivo, ou até ser definida em tempo de execução pelo usuário.

A distribuição da plataforma compilada é bastante simples. Ao compilar a plataforma com o compilador provido pela ferramenta *Google NaCL*, são gerados vários arquivos executáveis do tipo ”.nexe”para os vários tipos de plataformas do cliente (máquinas de 32 ou 64 bits). Estes arquivos devem ser colocados no mesmo diretório em que se encontra o arquivo HTML que vai carregar a aplicação e o próprio *browser Google Chrome* com o módulo *Native Client* ativado irá se encarregar de utilizar a versão adequada do executável, executando-o de maneira totalmente transparente.

6 CONCLUSÕES

A plataforma desenvolvida permite que aplicações com agentes BDI sejam criadas seguindo a linguagem de descrição de agentes 3APL-M, dando a oportunidade de implementar sistemas multiagentes na Web com maior facilidade para os desenvolvedores Web. Do ponto de vista de um desses desenvolvedores, usuários da plataforma, o conhecimento necessário para criar uma nova aplicação é de criação de páginas para a interface com o usuário final (HTML, CSS e *JavaScript*) e da linguagem de descrição de agentes 3APL-M.

As restrições para utilização da plataforma são o uso do *browser Google Chrome*, com o *Google NaCl* ativado, que atualmente é o único *browser* que suporta essa tecnologia, permitindo seu uso em sistemas 32 ou 64 bits e disponível para as plataformas Mac OS X, Linux e Windows.

6.1 LIMITAÇÕES ENCONTRADAS

Atualmente a tecnologia *Google NaCl* não possui implementada a função *ExecuteString()*, que poderia ser chamada de dentro do módulo para executar código *JavaScript* na página da aplicação. No momento, a única forma de comunicação partindo do módulo em C ou C++ com o *browser* é através do retorno da função executada, limitando as possibilidades de um agente.

6.2 CONTRIBUIÇÕES

- Uma versão extendida da plataforma 3APL-M na linguagem C++;
- Plataforma integrada para desenvolvimento de agentes 3APL-M para aplicações Web;

6.3 TRABALHOS FUTUROS

- Concluir a implementação do sistema de comunicação entre agentes, com possíveis modificações sobre a comunicação da linguagem 3APL-M;

- Melhorar outros aspectos da linguagem 3APL-M, como permitir uso de funções para cálculo de custo de planos;
- Adicionar a chamada de funções *JavaScript* para atuadores e sensores dos agentes (atualmente limitado por restrições da tecnologia *Google NaCl*).

REFERÊNCIAS

ALVARES, L. O.; SICHMAN, J. *Introdução aos Sistemas Multiagentes*. 1997. Jornada de atualização em Informática, 16.; Congresso da SBC, 17.; Brasília.

ARDISSONO, L.; GOY A.; PETRONE G.; SEGNAN M. A multi-agent infrastructure for developing personalized web-based systems. *ACM Trans. Internet Technol.*, ACM, New York, NY, USA, v. 5, n. 1, p. 47–69, 2005. ISSN 1533-5399.

BELLIFEMINE, F.; POGGI, A.; RIMASSA, G. Jade: a fipa2000 compliant agent development environment. In: *AGENTS '01: Proceedings of the fifth international conference on Autonomous agents*. New York, NY, USA: ACM, 2001. p. 216–217. ISBN 1-58113-326-X.

BIRUKOV, A.; BLANZIERI, E.; GIORGINI, P. Implicit: an agent-based recommendation system for web search. In: *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*. New York, NY, USA: ACM, 2005. p. 618–624. ISBN 1-59593-093-0.

BITTENCOURT, G. *Inteligência Artificial: Ferramentas e teorias.* [S.l.]: Editora UFSC, 2001.

BORDINI, R. H.; WOOLDRIDGE, M.; HÜBNER, J. F. *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*. [S.l.]: John Wiley & Sons, 2007. ISBN 0470029005.

BORDINI, R. H.; DASTANI M.; DIX J.; FALLAH-SEGHROUCHNI A. E. (Ed.). *Multi-Agent Programming: Languages, Platforms and Applications*. [S.l.]: Springer, 2005. (Multiagent Systems, Artificial Societies, and Simulated Organizations, v. 15). ISBN 0-387-24568-5.

BOUDRIGA, N.; OBAIDAT, M. S. Intelligent agents on the web: A review. *Computing in Science and Engineering*, IEEE Computer Society, Los Alamitos, CA, USA, v. 6, p. 35–42, 2004. ISSN 1521-9615.

BRATMAN, M. E. Intention and means-end reasoning. *Philosophical Review*, v. 90, n. 2, p. 252–265, 1981.

BRATMAN, M. E. *Intention, Plans, and Practical Reason*. [S.l.]: Cambridge University Press, 1999. ISBN 1575861925.

BRAUBACH, L.; POKAHR, E.; LAMERSDORF, W. Jadex: A bdi agent system combining middleware and reasoning. In: *Ch. of Software Agent-Based Applications, Platforms and Development Kits*. [S.l.]: Birkhaeuser, 2005. p. 143–168.

BRENNER, W.; WITTIG, H.; ZARNEKOW, R. *Intelligent Software Agents: Foundations and Applications*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1998. ISBN 3540634118.

CLOCKSIN, W. F.; MELLISH, C. S. *Programming in Prolog*. Springer-Verlag, 2003. ISBN 9783540006787. Disponível em: <http://books.google.com/books?id=VjHk2Cjrti8C>.

COULOURIS, G. F.; DOLLIMORE, J. *Distributed systems: concepts and design*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1988. ISBN 0-201-18059-6.

DASTANI, M. *3APL Platform User Guide www.cs.uu.nl/3apl/download/java/userguide.pdf*. 2006.

DASTANI, M.; RIEMSDIJK, B. van; MEYER, J. Ch. Programming multi-agent systems in 3APL. In: BORDINI, Rafael P. et al. (Ed.). *Multi-Agent Programming*. Springer, 2005, (Multi-agent Systems, Artificial Societies, and Simulated Organizations, v. 15). p. 39–67. ISBN 978-0-387-24568-3. Disponível em: <http://www.springerlink.com/content/t76827730l40wx0m/>.

ETZIONI, O.; WELD, D. S. Intelligent agents on the internet: Fact, fiction, and forecast. *IEEE Expert: Intelligent Systems and Their Applications*, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 10, p. 44–49, 1995. ISSN 0885-9000.

FENG, N.; DONG Y.; ZHANG A.; GUO Z. Research on intelligent web-learning based on multi-agents. In: *ICIC'07: Proceedings of the intelligent computing 3rd international conference on Advanced intelligent computing theories and applications*. Berlin, Heidelberg: Springer-Verlag, 2007. p. 190–195. ISBN 3-540-74170-4, 978-3-540-74170-1.

GAMMA, E.; HELM R.; JOHNSON R.; VLISSIDES J. M. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1. ed. [S.l.]: Addison-Wesley Professional, 1994. Hardcover. ISBN 0201633612.

GASSER, L. G.; BOND, A. H. *Readings in distributed artificial intelligence / edited by Alan H. Bond and Les Gasser*. [S.l.]: M. Kaufmann, San Mateo, Calif. :, 1988. xvii, 649 p. : p. ISBN 093461363.

GEORGEFF, M. P.; LANSKY, A. L. *Reactive Reasoning and Planning*. [S.l.]: American Association of Artificial Intelligence, 1987. 677–682 p.

GOOGLE-CODE. Acessado em Maio de 2011. Google Native Client. Disponível em: <http://code.google.com/p/nativeclient/>.

HUBER, M. J.; LEE J.; KENNY P.; DURFEE E. H. *UM-PRS V2.9 Programmer and User Guide*. [S.l.]: Artificial Intelligence Laboratory, The University of Michigan, 1993.

KOCH, F. *3APL-M: Platform for Lightweight Deliberative Agents www.cs.uu.nl/3apl-m/*. 2005.

MINOTTIE, M. *JaCa-Web, Progettare e Programmare Applicazioni Web 2.0 ad Agenti*. 2010.

RAO, A. S. Agentspeak(l): Bdi agents speak out in a logical computable language. In: . [S.l.]: Springer-Verlag, 1996. p. 42–55.

RAO, A. S.; GEORGEFF, M. P. Bdi agents: From theory to practice. In: *IN PROCEEDINGS OF THE FIRST INTERNATIONAL CONFERENCE ON MULTI-AGENT SYSTEMS (ICMAS-95)*. [S.l.]: MIT Press, 1995. p. 312–319.

RUSSELL, S.; NORVIG, P. *Artificial Intelligence: A Modern Approach (Second Edition)*. [S.l.]: Prentice Hall, 2003.

SHOHAM, Y.; LEYTON-BROWN, K. *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge, UK: Cambridge University Press, 2009. ISBN 978-0-521-89943-7.

STERLING, L.; SHAPIRO, E. Y. *The art of Prolog: advanced programming techniques*. MIT Press, 1994. (Journal of functional and logic programming). ISBN 9780262193382. Disponível em: <http://books.google.com/books?id=w-XjuvpOrjMC>.

WOOLDRIDGE, M. J. *Reasoning about Rational Agents*. [S.l.]: The MIT Press, Cambridge, Massachusetts, 2000. ISBN 0262232138.