UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

IVAN MEDEIROS MONTEIRO

TWProxy: Uma Ferramenta de Teamwork para Ambientes Dinâmicos com Requisitos de Tempo-Real

Dissertação apresentada como requisito parcial
para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Dr. Luís Otávio Campos Álvares
Orientador

Porto Alegre, fevereiro de 2009

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Monteiro, Ivan Medeiros

TWProxy: Uma Ferramenta de Teamwork para Ambientes Dinâmicos com Requisitos de Tempo-Real / Ivan Medeiros Monteiro. – Porto Alegre: PPGC da UFRGS, 2009.

89 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2009. Orientador: Luís Otávio Campos Álvares.

1. Coordenação em sistemas multiagentes. 2. Sistemas multiagentes em ambientes dinâmicos. 3. Teamwork. I. Álvares, Luís Otávio Campos. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitor de Pós-Graduação: Prof. Aldo Bolten Lucion
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenador do PPGC: Prof. Álvaro Freitas Moreira
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

“Everything should be made as simple as possible,
but not simpler.”
— ALBERT EINSTEIN

AGRADECIMENTOS

Sou imensamente grato à minha família por todo apoio durante este trabalho, em especial à minha esposa Bianca Monteiro, que além de dar todo o suporte emocional, dedicou-se bastante para as revisões dos meus textos. Ao meu orientador, conselheiro e amigo Luís Otávio Álvares, por ter sido efetivamente presente em todas as fases deste trabalho. A Vânia Bogorny, pelas sofridas revisões dos artigos produzidos no decorrer deste trabalho. A Andrey Palma, por me viciar no chimarrão logo na minha chegada à UFRGS. A Filipo Perotto, por todo apoio moral e compartilhamento de experiência. Aos voluntários, Fábio Cecin, Carlos Bezerra, Gustavo Silva, Daniel Epstein, Gabriel Hernandez, Marcelo Regina, Matheus Proença, Matheus Vogel, Bruno Guedes, Bruno Jurkoski, Alexandre Lorenzatti, Sandro Fiorini, Bárbara Gonçalvez, Diego Pereira, Eduardo Dias, Jefferson Bosa, Guilherme Zadrini, Leonardo Couto, Ramon Medrado, Germano Carniel e Leonardo Costa, que contribuíram com a execução dos experimentos envolvendo times de humanos. Ao CNPq pelo suporte financeiro.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS 7

LISTA DE FIGURAS 8

LISTA DE TABELAS 9

RESUMO 10

ABSTRACT 11

1 INTRODUÇÃO 12

2 TEAMWORK 14
2.1 Exemplo do comboio 14
2.2 Exemplo dos helicópteros militares 15
2.3 Modelos teóricos de teamwork 16
2.3.1 Joint Intentions 16
2.3.2 Shared Plans 19
2.4 Ferramentas de teamwork 20
2.4.1 Infraestrutura RETSINA 21
2.4.2 STEAM Framework 23
2.4.3 Machinetta Proxy 25

3 TWPROXY E O MODELO DE TEAMWORK PROPOSTO 29
3.1 Descrição de planos e crenças 32
3.1.1 Linguagem 32
3.1.2 Arquivo completo de exemplo 35
3.2 Comunicação 37
3.3 Coordenação 38
3.3.1 Escolha do líder 38
3.3.2 Processo de alocação de papéis 39
3.4 Interface com o agente 42

4 EXPERIMENTOS E AVALIAÇÕES 43
4.1 Eficiência da alocação de papéis 43
4.2 Avaliação do período de inconsistência de time 43
4.3 Eficiência de teamwork 46
4.3.1 TWProxy-Team vs UTBots-Average 50
4.3.2 TWProxy-Team vs UTBots-Experienced 51

4.3.3 Machinetta-Team vs UTBots-Average 52
4.3.4 TWProxy-Team vs Machinetta-Team 53
4.3.5 TWProxy-Team vs Human-Team 54

5 CONSIDERAÇÕES FINAIS 57

REFERÊNCIAS 59

APÊNDICE A ARQUITETURA DOS AGENTES PARA UT2004 63
A.1 Arquitetura 63
A.2 Framework 66
A.2.1 Comunicação com jogo 66
A.2.2 Implementação da Arquitetura 66

APÊNDICE B PERCEPÇÕES E AÇÕES COM GAMEBOTS 69

APÊNDICE C RESULTADOS DAS PARTIDAS 77

APÊNDICE D ALGORITMO HÚNGARO 83

LISTA DE ABREVIATURAS E SIGLAS

UT2004	Unreal Tournament 2004
NRW	Normalized Relative Weight
PIT	Período de Inconsistência de Time
TOP	Team-Oriented Programming
JI	Joint Intentions
ABNT	Associação Brasileira de Normas Técnicas
IAF	Indigente Agent Framework
CTF	Capture The Flag

LISTA DE FIGURAS

Figura 2.1: Simulação militar de ataque de helicópteros. 15
Figura 2.2: Infraestruta multiagente RETSINA e infraestrutura de agente individual. 22
Figura 2.3: Hierarquia de operadores para o domínio de ataque aéreo 25
Figura 2.4: Arquitetura do *Machinetta Proxy*. 27

Figura 3.1: Modelo de *teamwork* baseado em *proxy*. 29
Figura 3.2: *TWProxy* organização. 31
Figura 3.3: Comunicação inter-*proxy*. 38
Figura 3.4: Processo de alocação de papéis. 39
Figura 3.5: Interface com o agente. 42

Figura 4.1: Média da percentagem do valor máximo alcançada pela heurística *NRW*. 44
Figura 4.2: Desvio do valor ótimo pela heurística *NRW*. 44
Figura 4.3: Papel Protect The Base. 47
Figura 4.4: Papel Capture The Flag. 48
Figura 4.5: Guassianas da soma das pontuações individuais nos combates entre *TWProxy-Team* e *UTBots-Average*. 51
Figura 4.6: Guassianas da soma das pontuações individuais nos combates entre *TWProxy-Team* e *UTBots-Experienced*. 52
Figura 4.7: Guassianas da soma das pontuações individuais nos combates entre *Machinetta-Team* e *UTBots-Average*. 53
Figura 4.8: Guassianas da soma das pontuações individuais nos combates entre *TWProxy-Team* e *Machinetta-Team*. 54
Figura 4.9: Guassianas da soma das pontuações individuais nos combates entre *TWProxy-Team* e *Human-Team*. 55

LISTA DE TABELAS

Tabela 4.1: Média e desvio padrão do Período de Inconsistência de Time. 45
Tabela 4.2: Resultado das pontuações de time e individuais nas partidas de *Capturar a Bandeira* entre o *TWProxy-Team* e o *UTBots-Average*. 50
Tabela 4.3: Resultado das pontuações de time e individuais nas partidas de *Capturar a Bandeira* entre o *TWProxy-Team* e o *UTBots-Experienced*. . . 52
Tabela 4.4: Resultado das pontuações de time e individuais nas partidas de *Capturar a Bandeira* entre o *Machinetta-Team* e o *UTBots-Average*. . . . 52
Tabela 4.5: Resultado das pontuações de time e individuais nas partidas de *Capturar a Bandeira* entre o *TWProxy-Team* e o *Machinetta-Team*. 53
Tabela 4.6: Resultado das pontuações de time e individuais nas partidas de *Capturar a Bandeira* entre o *TWProxy-Team* e o *Human-Team*. 54

RESUMO

Embora haja muitas pesquisas na área de *teamwork*, o desenvolvimento de times de agentes para ambientes complexos continua sendo um desafio, especialmente se esses ambientes possuem requisitos de tempo-real. Muitas ferramentas foram desenvolvidas, mas não existe uma bala de prata e aquelas mais gerais possuem sérios problemas com o requisito de tempo-real. Este trabalho introduz uma nova ferramenta baseada em *proxy*, usando a teoria de *Joint Intentions*, que ajuda agentes a serem membros de uma equipe dentro de um ambiente parcialmente observável, dinâmico, estocástico e com requisitos de tempo-real. Os experimentos de validação dessa ferramenta são executados no jogo de computador *Unreal Tournament 2004*, que fornece um ambiente com tais características.

Palavras-chave: Coordenação em sistemas multiagentes, sistemas multiagentes em ambientes dinâmicos, teamwork.

TWProxy: A Teamwork tool for Dynamic Environment with Real-Time Requirements

ABSTRACT

Although there are many researches in teamwork, the development of agent teams for complex environments still presents many challenges, especially if these environments have real-time requirements. Many tools have been developed, but there is not a silver bullet, and the most general tools have serious problems with real-time requirements. This paper introduces a new proxy-based tool, based on *Joint Intentions*, to help agents to be a teammate in partilly observable, dynamic and stochastic environments with real-time requirements. The validation experiments for this proxy were performed in the computer game Unreal Tournament 2004, that provides an environment with such features.

Keywords: Artificial intelligence, multiagents systems, teamwork.

1 INTRODUÇÃO

Diversos domínios de aplicação operam com requisitos de tempo real, sendo possível citar: jogos de computador, robótica móvel, automação industrial, controle aéreo, entre outros. Em jogos de computador, uma indústria que movimenta cifras bilionárias e já supera a indústria do cinema, a demanda por comportamentos convincentes tem crescido, forçando a diminuição da distância entre o comportamento humano e comportamento sintetizado (BYL, 2004). A exigência por realismos gráficos e comportamentos convincentes é cada vez maior. Quando se fala em jogos baseados em times, o contexto é ainda mais complexo, visto que freqüentemente os *bots*[1] não assumem papéis coerentes dentro do time. Assim, apesar dos avanços ocorridos na determinação do comportamento individual dos *bots*, pouco tem sido feito pelo comportamento social dos mesmos. Um paradigma de time, conhecido em sistemas multiagentes como *teamwork*[2], aborda este problema. No contexto deste trabalho, um time é um grupo de entidades autônomas, próativas e com capacidades de reflexão sobre as próprias habilidades e as habilidades do grupo, que compartilham um objetivo comum.

Teamwork tem emergido como o paradigma para coordenação de agentes de forma cooperativa em ambientes dinâmicos e tem se mostrado capaz de levar a um comportamento robusto e flexível (SYCARA; SUKTHANKAR, 2006). Entretanto, não existe uma solução geral para construção de times de agentes e cada novo domínio pode ter novos requisitos. Uma classe de aplicação que se mantém como desafio para a área de *teamwork* é aquela na qual o ambiente é altamente dinâmico e que requer um tempo de resposta muito curto.

Em ambientes dinâmicos, o estado do ambiente pode sofrer alteração entre o agente perceber tal ambiente e atuar sobre ele. Ou seja, o tempo que o agente demora para agir pode afetar a qualidade de suas ações. Isso se torna mais grave quando o ambiente exige uma reatividade muito alta, com intensas atualizações de seu estado. Em domínios altamente dinâmicos a manutenção da coerência de time é também dificultada, dado que a mudança de postura do time precisa acompanhar as mudanças de estado do ambiente.

Aplicações como jogos de computador, podem representar um ambiente com tais características, onde a formação de *teamwork* pode necessitar uma abordagem diferenciada (MONTEIRO; ALVARES, 2008). Nesses ambientes frequentemente é necessário sacrificar alguns aspectos gerais, como comunicação assíncrona ou decisões distribuídas, em favor do tempo de resposta. Assim, dependendo do domínio, ainda é possível formar

[1] Um personagem controlado por um agente artificial.

[2] Esforço cooperativo realizado por membros de uma equipe para alcançar um objetivo comum.

times robustos e coerentes em ambientes altamente dinâmicos.

Teamwork tem sido aplicado a diversos domínios, tais quais: resgate em desastres (NAIR et al., 2000), combate militar (HILL et al., 1997; JONES et al., 1999), *robocup soccer* (KAMINKA, 1999; MARSELLA et al., 1999, 2001), e colaboração entre humanos e agentes (SCERRI et al., 2003a). Um *teamwork* flexível, promovido por um modelo explícito, que define comprometimentos e responsabilidades para os membros do time, permite uma coordenação robusta, mantendo a coerência mesmo que haja falhas individuais e mudanças imprevisíveis no ambiente (SCHURR et al., 2005a).

Algumas teorias foram propostas para formalizar o comportamento de um grupo de agentes para agir como um time. Duas dessas teorias possuem importantes resultados em problemas reais. Elas são *Joint Intentions* (LEVESQUE; COHEN; NUNES, 1990) e *Shared Plans* (GROSZ; KRAUS, 1996). Ambas são descritas através de um formalismo lógico, mas com diferentes abordagens. *Joint Intentions* foca na junção dos estados mentais do time, enquanto *Shared Plans* foca nas intenções do agente voltadas para ações de colaboração ou ações conjuntas do grupo.

Baseadas nessas teorias surgiram ferramentas como *STEAM* (TAMBE, 1997a) e *Machinetta* (SCHURR et al., 2005a) que tiveram bons resultados em diversos domínios de ambiente complexo (HILL et al., 1997; TAMBE; ZHANG, 1998, 2000; MARSELLA et al., 2001; JONES et al., 1999; PYNADATH; TAMBE, 2003; SCERRI et al., 2003b; SCHURR et al., 2005b; SCERRI et al., 2003a,c; CHALUPSKY et al., 2001; SCERRI; PYNADATH; TAMBE, 2001, 2002; YEN et al., 2001). Entretanto, o *STEAM* foi desenvolvido usando uma versão antiga do *SOAR* (LAIRD; NEWELL; ROSENBLOOM, 1987), que teve mudança na sua linguagem desde então, e o *Machinetta*, que é um projeto derivado do *STEAM*, apresenta diversas limitações para ambientes altamente dinâmicos.

Nenhuma ferramenta atual oferece uma solução realmente geral para *teamwork*. A diversidade de domínios é o que, na prática, inviabiliza uma solução genérica que seja satisfatória para qualquer tipo de aplicação. Por isso, este trabalho foca na investigação e análise de *teamwork* em ambiente multiagente altamente dinâmico, estocástico, parcialmente observável, com o objetivo de propor uma ferramenta para o desenvolvimento de time de agentes.

A fim de superar essas limitações existentes, este trabalho introduz o *TWProxy*, uma nova ferramenta que habilita agentes individuais a integrarem times e agir de forma coerente em ambientes altamente dinâmicos com requisitos de tempo-real. Ele utiliza algumas das boas idéias apresentadas pelo *Machinetta*, contornando algumas de suas limitações e adicionando novas características.

O restante deste trabalho está organizado da seguinte forma: o Capítulo 2 apresenta os trabalhos de *teamwork* relacionados, com descrição dos problemas de *teamwork*, modelos teóricos e apresentação das principais ferramentas; o Capítulo 3 detalha o funcionamento do *TWProxy*, juntamente com seu modelo de *teamwork* e alguns exemplos sobre o seu uso; o Capítulo 4 relata os experimentos e avaliação da ferramenta aqui apresentada, mostrando as vantagens do *TWProxy* para ambientes altamente dinâmicos; no Capítulo 5 são feitas as considerações finais deste trabalho e apresentados alguns direcionamentos para trabalhos futuros; no Apêndice A são descritas a arquitetura e o *framework* utilizados para o desenvolvimento dos agentes para os experimentos; no Apêndice B são descritas as percepções e ações dos *bots* para o jogo UT2004; no Apêndice C são detalhados os

resultados dos experimentos com o jogo UT2004; e no Apêndice D é apresentada a implementação do algoritmo húngaro utilizado no *TWProxy*.

2 TEAMWORK

Diversos problemas multiagentes podem ser enquadrados como um problema de *teamwork*. Essencialmente, problemas de *teamwork* são aqueles onde um grupo de agentes compartilham um objetivo e precisam cooperar entre si agindo de forma coordenada para alcançar o objetivo. A diferença principal entre um problema de *teamwork* e de coordenação clássica é que em *teamwork* um objetivo compartilhado precisa ser alcançado com a cooperação dos agentes envolvidos. Para alguns domínios é possível ter um esquema de coordenação pré-combinado, assumindo a suposição da sala fechada(STONE; VELOSO, 1998), onde planos pré-definidos são executados pelo time após observar os ambientes e escolher o adequado. Em diversos ambientes dinâmicos, a suposição da sala fechada não é válida, o que torna a diferença entre *teamwork* e coordenação clássica ainda mais clara, quando é mais difícil manter a coerência de time. Abaixo são apresentados dois exemplos onde a coordenação clássica falha e é necessária a atuação em time.

2.1 Exemplo do comboio

Para ilustrar as diferenças entre *teamwork* e coordenação multiagente, Levesque (LEVESQUE; COHEN; NUNES, 1990) apresenta o exemplo de dirigir em comboio. Neste, existem dois agentes, A e B. O agente B deseja dirigir até sua casa mas não sabe o caminho, e A conhece o caminho para a casa de B. Assim, A pretende guiar B até um caminho que ele saiba seguir, cada um fazendo a sua parte. Neste exemplo, a coordenação clássica pode demonstrar vários pontos de falha de consistência.

Quando um agente acredita que seu objetivo ou foi alcançado, ou tornou-se impossível de alcançar, uma escolha racional seria abandonar o objetivo. Por exemplo, se o caminho que A conhecia está interditado e ele não conhece outro caminho que leve à casa de B, a intenção de guiar B deveria ser abortada. Neste caso, a coordenação não fornece nenhum compromisso entre A e B, não existindo nada que impeça que A simplesmente siga seu caminho. Se B percebe que conhece o caminho restante para sua casa, o objetivo para B foi satisfeito, então ele poderia seguir para casa sem considerar como A iria interpretar sua ação. Como A não tem consciência que o objetivo foi alcançado, continua tentando guiar B para o caminho de sua casa.

Até mesmo se um agente leva em consideração as ações do outro, a falta de coerência ainda poderia ocorrer. Se A faz um retorno, B poderia erroneamente concluir que A não está habilitado para acompanhá-lo e poderia executar outros planos para chegar em casa. Ou se B é forçado a parar, A poderia simplesmente interpretar que B sabe o caminho de

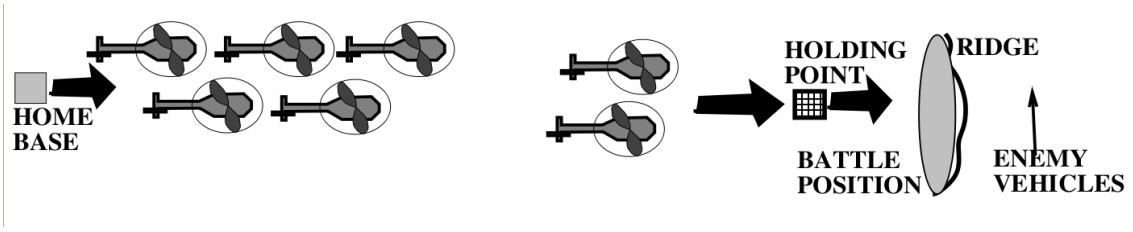

Figura 2.1: Simulação militar de ataque de helicópteros.

casa e continuar dirigindo sem ele.

2.2 Exemplo dos helicópteros militares

O exemplo dos helicópteros militares(JONES et al., 1999) envolve um caso de ataque como exemplificado na Figura 2.1. O grupo começa na base, onde o piloto comandante primeiro envia as ordens e instruções para os membros da equipe. O grupo processa as ordens e então segue até a posição de espera, próximo ao campo de batalha. Uma vez que o grupo alcança sua posição de espera especificada, ele aguarda enquanto os helicópteros de reconhecimento seguem ao campo de batalha. Baseados na comunicação entre os helicópteros de reconhecimento e o restante do grupo, os demais seguem para o campo de batalha a fim de realizar um ataque. Concluído o ataque, os helicópteros se reagrupam e voltam para base. Caso algum veículo inimigo seja encontrado pelo caminho em posição de ameaça, os demais são alertados para que seja feita uma nova rota que evite o veículo inimigo.

Nesse domínio haverá potenciais falhas de *teamwork* caso os membros da equipe não estejam comprometidos em manter a coerêcia do time. Abaixo são enumeradas possíveis falhas na coerência de time para o domínio anterior.

1. Após batalha com o inimigo, o comandante poderia retornar sozinho à base, abandonando os membros de sua equipe no campo de batalha.

2. Após a chegada do grupo no ponto de espera, um helicóptero de reconhecimento deveria se certificar de que a área atrás da colina estaria livre. Mas nessa verificação ele poderia ser abatido, deixando o resto da equipe esperando indefinidamente.

3. Um piloto, inesperadamente, processa as ordens iniciais antes dos outros. Ele então segue sozinho para o campo de batalha enquanto o restante do time continua na base.

4. Após helicópteros serem abatidos, apenas um de escolta chega na posição de espera. O mesmo espera indefinidamente os demais helicópteros para seguir em frente.

5. Quando a ordem inicial falha em atribuir o papel de escolta para um membro do time, o grupo inteiro espera indefinidamente quando chegar no ponto de espera.

6. As instruções enviadas pelo comandante para alguns membros foram perdidas porque o comandante enviou a mensagem quando os membros da equipe estavam ocupados com outra tarefa. Neste caso, estes membros estão impossibilitados de selecionar as ações apropriadas.

7. Enquanto o grupo desvia de um veículo inimigo, inesperadamente um dos helicópteros destrói o veículo. Entretanto, os demais pilotos não são informados, assim uma rota alternativa é planejada desnecessariamente.

8. Em um caso extremo, quando a munição de todos os membros do time acaba, o grupo falha em inferir que a missão não poderia continuar.

9. Dois grupos de helicópteros acidentalmente usam o mesmo canal de rádio, levando à interferência e perda da mensagem inicial enviada pelo comandante. O grupo espera indefinidamente na base.

2.3 Modelos teóricos de teamwork

Trabalhos teóricos em *teamwork* (LEVESQUE; COHEN; NUNES, 1990) (GROSZ; KRAUS, 1996) (GROSZ; KRAUS, 1999) de agentes formalizam o comportamento de time e definem alguns pré-requisitos para a atuação coerente em equipe, tais como: os agentes precisam compartilhar os objetivos que eles querem alcançar, um plano de tudo que eles seguem juntos, além de conhecimentos do ambiente no qual operam; eles também necessitam compartilhar as intenções de executar um plano para alcançar um objetivo comum; os membros do time precisam ser conscientes das suas capacidades e de como podem preencher os papéis necessários para os planos de alto nível do time; membros do time devem estar aptos para monitorar seus próprios progressos voltados aos objetivos do time, as atividades dos outros membros e as *Joint Intentions* do time(LEVESQUE; COHEN; NUNES, 1990). Muitos sistemas têm sido desenvolvidos usando essas idéias básicas de *teamwork*.

2.3.1 Joint Intentions

Segundo *Joint Intentions* (LEVESQUE; COHEN; NUNES, 1990), ações conjuntas de um time não consistem apenas de ações simultâneas e individualmente coordenadas. Para atuar em conjunto, um time deve ter consciência do estado da equipe e dos esforços do grupo como um todo. Por exemplo, no tráfego de automóveis, onde os motoristas atuam simultaneamente e são coordenados pelos sinais e leis de trânsito, não existe de fato *teamwork* envolvido. Mas se um grupo de motoristas decide fazer alguma coisa em conjunto, como por exemplo, dirigir em comboio, irá parecer que o grupo age como um único agente com crenças, objetivos e intenções próprias. Baseado nisso, *Joint Intentions* apresenta um modelo formal das propriedades mentais de um grupo de agentes que deseja executar ações conjuntas.

O formalismo de *Joint Intentions* foca na junção de estados mentais de membros de um time. Nele, para que um time faça uma ação em conjunto é necessário que seus membros se comprometam a completar a ação do grupo, enquanto mutuamente acreditam que é possível fazê-la. Além de se comprometer a realizar a tarefa, os membros de um time se comprometem a tornar de conhecimento comum o fato de um objetivo ser concluído, tornar-se impossível de se atingir ou deixar de ser importante.

Em *Joint Intentions*, assume-se que os agentes são modelados em um ambiente multi-agente dinâmico, sem possuir uma crença completa, nem necessariamente correta, do mundo e dos demais agentes. Eles possuem objetivos modificáveis e suas ações são falí-

veis. Por simplicidade é também assumido que uma vez o agente venha a pensar que o objetivo não é mais alcançável, ele nunca muda de idéia, e que sempre é possível alcançar a crença mútua.

2.3.1.1 Individual Commitment

O formalismo de *Joint Intention* é expresso em uma linguagem modal de primeira-ordem com o conectivo de igualdade e os operadores proposicionais. Nesta sub-seção são apresentadas as definições necessárias para compor as definições de comprometimento mútuo do time. A seguir são descritos os operadores da linguagem utilizada.

- $BEL(x,p)$, o agente x acredita em p.
- $GOAL(x,p)$, o agente x tem p como objetivo.
- $MB(x,y,p)$, os agentes x e y acreditam mutuamente em p.
- $AGT(x_1,\ldots,x_n,e)$, os agentes x_1, ..., x_n são os únicos agentes para a seqüência de eventos e.
- $e_1 \leq e_2$, e_1 é uma subseqüência anterior a e_2.
- $HAPPENED(a)$, a aconteceu.
- $HAPPENING(a)$, a está acontecendo.
- $HAPPENS(a)$, a irá acontecer.
- $a;b$, composição de ações.
- $a|b$, escolha não determinística.
- $a||b$, ocorrência concorrente de a e b.
- $p?$, teste de ação.
- a^*, repetição da ação a.

Dentro da modelagem é assumido que cada agente tem um conhecimento perfeito sobre suas crenças e objetivos, que as crenças são consistentes, e que os objetivos são consistentes entre si e com a crença.

Para simplificar a notação, são feitas as seguintes abreviações sintáticas para as ações:

- $DONE(x_1,\ldots,x_n,a) \stackrel{def}{=} HAPPENED(a) \wedge AGT(x_1,\ldots,x_n,e)$
- $DOING(x_1,\ldots,x_n,a) \stackrel{def}{=} HAPPENING(a) \wedge AGT(x_1,\ldots,x_n,e)$
- $DOES(x_1,\ldots,x_n,a) \stackrel{def}{=} HAPPENS(a) \wedge AGT(x_1,\ldots,x_n,e)$

Os operadores modais $\Diamond$ *eventually*, $\Box$ *always* e *until* são definidos a seguir com base nos predicados e operadores acima:

- $\Diamond p \stackrel{def}{=} \exists e HAPPENS(e;p?)$,
  p é verdade em algum ponto do futuro.

- $\Box p \stackrel{def}{=} \neg\Diamond\neg p$,
  p é sempre verdade a partir do momento atual.

- $UNTIL(p,q) \stackrel{def}{=} \forall c HAPPENS(c;\neg q?) \rightarrow \exists a(a \leq c) \wedge HAPPENS(a;p?)$,
  p é verdade até que q seja verdade.

Para concluir as definições de compromissos e intenções individuais, são definidos *PGOAL* e *INTEND*. *PGOAL* representando um objetivo individual persistente, onde o agente acredita que o seu objetivo ainda não foi atingido e que é possível atingí-lo. Assim, ele continua no objetivo enquanto ainda for possível ou relevante continuar. O *INTEND* representa a intenção do agente em executar uma ação, enquanto mantém um objetivo persistente que inclui tal ação e acredita que a está excutando.

- $PGOAL(x,p,q) \stackrel{def}{=} BEL(x,\neg p) \wedge GOAL(x,\Diamond p) \wedge UNTIL([BEL(x,p) \vee BEL(x,\Box\neg p) \vee BEL(x,\neg q)], GOAL(x,\Diamond p))$
  Onde x é o agente, p é o objetivo de x e q é o critério que torna p irrelevante.

- $INTEND(x,a,q) \stackrel{def}{=} PGOAL(x, DONE(x, UNTIL(DONE(x,a), BEL(x, DOING(x,a)))?;a), q)$,
  Onde x é o agente, a é a ação que o agente pretende fazer e q é o critério que torna irrelevante executar a ação.

2.3.1.2 Joint Commitment

Uma generalização do *PGOAL* para o caso onde o grupo atua como um agente é dada pelo predicado $JPG(x,y,p,q)$. Para se definir o *JPG* é necessário antes definir o objetivo mútuo *MG* e o objetivo fraco *WG*. No objetivo mútuo, os agentes envolvidos devem acreditar que todos possuem o mesmo objetivo, e no objetivo fraco o agente envolvido continua com seu objetivo enquanto acredita que ele não foi alcançado, mas caso perceba que ele foi alcançado ou que se tornou inviável, torna conhecimento mútuo o novo fato.

- $MG(x,y,p) \stackrel{def}{=} MB(x,y,GOAL(x,\Diamond p) \wedge GOAL(y,\Diamond p))$
  Onde x e y são agentes com um mesmo objetivo p

- $WG(x,y,p) \stackrel{def}{=} [\neg BEL(x,p) \wedge GOAL(x,\Diamond p)] \vee [BEL(x,p) \wedge GOAL(x, \Diamond MB(x,y,p))] \vee [BEL(x,\Box\neg p) \wedge GOAL(x,\Diamond MB(x,y,\Box\neg p))]$
  Onde x e y são agentes com um mesmo objetivo p

- $JPG(x,y,p,q) \stackrel{def}{=} MB(x,y,\neg p) \wedge MG(x,y,p) \wedge UNTIL([MB(x,y,p) \vee MB(x,y,p) \vee MB(x,y,\neg q)], MB(x,y,WG(x,y,p) \wedge WG(y,x,p)))$
  Onde x e y são agentes com um objetivo p e com uma cláusula de relevância q

2.3.1.3 Joint Action

Dada a noção de comprometimento em conjunto é possível definir *Joint Intention* (*JI*) como uma generelização da intenção individual, que se resume em um comprometimento em conjunto para realizar uma ação enquanto todos, mutuamente, acreditam que estão executando a ação.

- $JI(x,y,a,q) \stackrel{def}{=} JPG(x,y,$
  $DONE(x,y,UNTIL(DONE(x,y,a),MB(x,y,DOING(x,y,a)))?;a),q)$
  Onde x e y são agentes envolvidos na ação em grupo, a é a ação executada em grupo e q a cláusula que torna a execução da ação a irrelevante.

2.3.2 Shared Plans

Em contraste com *Joint Intentions*, o conceito de *SharedPlans* (GROSZ; KRAUS, 1996) não é baseado na junção de atitudes mentais. Ao invés disso, *SharedPlans* remete ao conceito de atitudes intencionais, *intending-that*, o qual é similar a uma ação normal de um agente. Entretanto, um *intending-that* de um agente individual é voltado para ações de colaboração ou para uma ação em conjunto do grupo. *Intending-that* é definido por um conjunto de axiomas que guia um indivíduo a realizar ações, incluindo ações de comunicação, que permitem membros do time, sub-time ou o time executar a tarefa atribuída (GROSZ; KRAUS, 1996).

Um *SharedPlans* pode ser representado por um *full SharedPlans* (FSP) ou por um *partial SharedPlans* (PSP). Um FSP para realizar uma ação α representa uma situação onde todos os aspectos de uma atividade conjunta α é completamente determinada. Isto inclui crença mútua e acordos em uma receita completa R_α para fazer a ação α. R_α é uma especificação de um conjunto de ações β_i, a qual, quando executada sobre as restrições especificadas, constitui a execução de α. $FSP(P,GR,\alpha,T_p,T_\alpha,R_\alpha)$ denota o plano P de um grupo GR no tempo T_p para fazer a ação α no tempo T_α usando a receita R_α. De forma resumida, $FSP(P,GR,\alpha,T_p,T_\alpha,R_\alpha)$ é verdade se e somente se as seguintes condições são satisfeitas:

1. Todos os membros do grupo GR acreditam mutuamente que eles pretendem que a proposição $Do(GR,\alpha,T_\alpha)$ seja verdade, dado que GR faz α no tempo T_α.
2. Todos os membros de GR mutuamente acreditam que R_α é receita de α
3. Para cada passo de β_i em R_α:
   - Um sub-grupo $GR_k(GR_k \subseteq GR)$ tem um FSP para β_i, usando a receita $R_\beta i$. (GR_k pode ser apenas individual, no caso ele deve ter um *full individual plan*, uma analogia ao FSP de forma individual.)
   - Outro membro de GR acredita que existe uma receita tal que GR_k pode levar a um β_i e ter um FSP para β_i (mas outros membros podem não conhecer $R_\beta i$)
   - Outros membros de GR pretendem que GR_k possa levar a β_i usando alguma receita.

A teoria de *SharedPlans* se propõe a descrever uma rede inteira de crenças e intenções de time quando engajado em *teamwork*. Nesta tentativa, um FSP representa um caso limite. Normalmente, quando engajado em uma atividade de time, um time só tem um *partial SharedPlan* (PSP). O PSP é uma foto do estado mental do time em uma situação particular em seu *teamwork*, e comunicação e planejamento são freqüentemente utilizados para completar as condições de um FSP (embora, em um ambiente dinâmico, o time possa nunca realmente formar um FSP). Abaixo seguem três exemplos nos quais podem existir um PSP.

- A receita R_α pode ser apenas parcialmente especificada. Como o time decide reativamente o próximo passo baseado no contexto e na situação atual, poderiam ser consideradas evoluções das receitas ao longo do tempo, principalmente em ambientes dinâmicos. De acordo com a teoria de *SharedPlans*, membros do time devem chegar a uma crença mútua nos próximos passos β_i. Para cada passo β_i na receita, o subgrupo relevante deve formar um *SharedPlan*.

- A alocação de tarefas do time pode não ser conciliada. Por exemplo, o agente ou o grupo para executar uma tarefa pode não ser determinado. Nesta situação, membros do time pretendem que exista algum indivíduo ou sub-grupo para fazer a tarefa. Dentre as ações consideradas como um resultado de um *intending-that*, indivíduos podem se voluntariar para executar a tarefa não conciliada ou persuadir/ordenar outros a pegar a tarefa.

- Indivíduos ou sub-grupos podem não ter atingido uma crença mútua apropriada para formação de um FSP, levando à comunicação dentro do time. Essa comunicação também pode surgir devido às atitudes de *intenting-that* dos agentes, voltadas aos objetivos do time e às atividades dos membros do time. Por exemplo, um membro do time pretende que o time faça uma ação β_i, e acredita que a comunicação de alguma informação particular irá habilitar o time a fazer β_i. Isso irá levá-lo a comunicar tal informação ao grupo, desde que a comunicação não conflite com os compromissos anteriores.

2.4 Ferramentas de teamwork

Baseadas em *Joint Intentions* e *SharedPlans* surgiram várias ferramentas que auxiliam na manutenção da coerência de *teamwork*, tais como: *GRATE** (JENNINGS, 1995), *COLLAGEN* (RICH; SIDNER, 1997), *STEAM* (TAMBE, 1997a), *RETSINA* (SYCARA et al., 2003), *MONAD* (VU et al., 2003), *Machinetta* (SCERRI et al., 2003c) e *TWProxy* (MONTEIRO; ALVARES, 2008).

*GRATE** é uma versão extendida do *GRATE*(*Generic Rules and Agent model Testbed Environment*), um sistema integrado de propósito geral que contém conhecimento genérico sobre cooperação embutido. O *GRATE** é baseado em *Joint Responsibility*, que por sua vez é baseado em *Joint Intentions*. Ele propõe uma arquitetura integrada de agente com duas camadas, uma do sistema de domínio e outra de controle e cooperação, tendo a sua principal aplicação no domínio da indústria.

O *COLLAGEN* é uma ferramenta feita em *Common Lisp* e baseada em *SharedPlan* que incorpora a idéia do discurso colaborativo (RICH; SIDNER, 1997). Ele foi inicialmente desenvolvido para uma aplicação simples de escolha de rotas aéreas e mantém o foco na interação entre humano e agentes. O *COLLAGEN* segue o paradigma de que a colaboração de agentes com humanos deve ser governada pelos mesmos princípios que seguem as interações entre as pessoas.

RETSINA é uma infraestrutura de sistemas multiagentes que utiliza mecanismo de raciocínio de time baseado em *SharedPlans*, com os objetivos de traçar tarefas interdependentes para membros do time, reconhecer e reportar conflitos, propor soluções para resolver os conflitos e monitorar a performance do time.

O *STEAM* é um modelo genérico para implementação de *teamwork*. Foi desenvolvido sobre a plataforma *SOAR*(LAIRD; NEWELL; ROSENBLOOM, 1987) e é baseado nas teorias de *Joint Intentions* e *SharedPlans*. Diversos resultados demonstram o seu sucesso na composição de times de agentes em ambientes dinâmicos (TAMBE, 1997b,a; HILL et al., 1997; TAMBE; ZHANG, 1998; JONES et al., 1999; TAMBE; ZHANG, 2000; MARSELLA et al., 2001).

O *MONAD* (VU et al., 2003) define toda uma arquitetura de controle multiagente inspirada no funcionamento do *STEAM*. Ele integra o projeto do time baseado em programação de *script off-line* com um motor de coordenação em tempo de execução. Seu principal avanço em relação ao *STEAM* é permitir que o projetista defina diferentes métodos de arbitragem multiagente, além de fornecer um conjunto de ferramentas gráficas para o desenvolvimento de sistemas multiagentes.

Machinetta é uma evolução do modelo de *teamwork* do *STEAM*. Implementado em Java, ele segue uma abordagem baseada em *proxy*, que visa habilitar agentes não comprometidos socialmente ao comportamento social. Introduz um novo algoritmo de alocação de papéis e tem como característica importante a cooperação entre robôs, agentes e humanos através do ajuste de autonomia das entidades.

Devido às limitações das ferramentas acima em fornecer suporte de *teamwork* para ambientes altamente dinâmicos, este trabalho apresenta o *TWProxy*. Baseado em *Joint Intentions*, o *TWProxy* utiliza algumas características do *Machinetta*, superando suas principais limitações em atender a requisitos de tempo-real e introduzindo novas características úteis para domínios altamente dinâmicos. A seguir, são descritas mais detalhadamente três dessas ferramentas.

2.4.1 Infraestrutura RETSINA

RETSINA é uma infraestrutura aberta de sistema multiagente que suporta comunidades de agentes heterogêneos. O sistema *RETSINA* foi implementado com a idéia de que o agente deveria formar uma comunidade de pontos que se engajam em uma relação ponto a ponto. Qualquer estrutura de coordenação na comunidade de agentes deve emergir da relação entre os agentes ao invés de imposta pela infraestrutura. Seguindo essa premissa, *RETSINA* não emprega controle centralizado em sistemas multiagentes, em vez disso ele implementa serviços de infraestrutura distribuída que facilitam a relação entre agentes em vez de gerenciá-los.

O *RETSINA* é definido como um conjunto de serviços, convenções e conhecimentos que dá suporte às interações sociais complexas. Os agentes precisam dos serviços para tornarem-se aptos a encontrar os outros em um ambiente aberto, para se comunicar e para garantir que as restrições de segurança serão satisfeitas. Convenções, como linguagem de comunicação de agentes e políticas de conversação, são a base para alcançar um acordo sobre o que os agentes estão fazendo e o que eles estão alcançando. O agente precisa também do conhecimento de como usar a infraestrutura, as linguagens de comunicação e protocolos, bem como uma ontologia comum para ter uma participação efetiva na comunidade.

A organização da infraestrutura de sistemas multiagentes *RETSINA* é mostrada na Figura 2.2. Ela apresenta como os vários componentes são organizados. O restante desta seção

RETSINA MAS INFRASTRUCTURE	INDIVIDUAL AGENT INFRASTRUCTURE IN RETSINA
MAS INTEROPERATION RETSINA-OAA Interoperator	
CAPABILITY TO AGENT MAPPING Matchmaker	**CAPABILITY TO AGENT MAPPING** Matchmaker Module
NAME TO LOCATION MAPPING ANS	**NAME TO LOCATION MAPPING** ANS Module
SECURITY Certificate Authority Cryptography Services	**SECURITY** Security Module private/public Keys
PERFORMANCE SERVICES Failure Monitoring	**PERFORMANCE SERVICES** Self Monitoring Cloning
MAS MANAGEMENT SERVICES Logger ActivityVisualizer Launcher	**MANAGEMENT SERVICES** Logger Module
ACL INFRASTRUCTURE Public Ontology Protocols Servers	**ACL INFRASTRUCTURE** ACL Parser Private Ontology Protocol Engine
COMMUNICATION INFRASTRUCTURE Discovery Message Transfer	**COMMUNICATION MODULES** Discovery Module RETSINA Communicator
OPERATING ENVIRONMENT Machines, OS, Network Multicast Transport Layer: TCP/IP, Wireless, Infrared, SSL	

Figura 2.2: Infraestruta multiagente RETSINA e infraestrutura de agente individual.

descreve cada um desses componentes.

- **Ambiente de operação:** O *RETSINA* é independente da plataforma na qual os componentes de infraestrutura e os agentes executam. Ele automaticamente lida com os diferentes tipos de camadas de transporte.

- **Infraestrutura de comunicação:** O *RETSINA* é baseado em dois tipos de canais de comunicação. Um provê transferência de mensagens por comunicação direta ponto a ponto entre os agentes; o outro é baseado em *multicast* usada para descobrir processos que levem o agente a achar os componentes de infraestrutura.

- **Infraestrutura de linguagem de comunicação de agente:** A troca de mensagens entre agentes no RETSINA é feita em KQML(FININ et al., 1994). As mensagens possuem dois componentes: a especificação do conteúdo da mensagem e o envelope da mensagem. No envelope são especificadas informações como o remetente, o destinatário, o histórico da conversação, a ontologia e a linguagem utilizadas no conteúdo. O formato do envelope é fixado pelo *RETSINA*, entretanto nenhuma suposição é feita quanto ao conteúdo da mensagem.

- **Gestão de serviços:** O gerenciamento de aplicações é uma tarefa complexa que fica mais difícil com o tamanho da aplicação e o aumento do número de máquinas e agentes. Ele inclui três componentes de gerenciamento: o *Logger*, *Activity Visualizer* e *Laucher*, que formam um conjunto de ferramentas para ajudar a depurar, monitorar e lançar alguma aplicação.

- **Serviços de execução:** O *RETSINA* provê serviços de monitoramento na simula-

ção, *check-pointing* e *roll-back* distribuídos.

- **Segurança:** Como o *RETSINA* é um sistema aberto, agentes desconhecidos e possivelmente não confiáveis podem entrar a qualquer momento. Esses agentes podem causar danos ao sistema de várias formas: eles podem espionar outros agentes, roubar informações e danificar o conteúdo dos componentes de infraestrutura. A infraestrutura de segurança do *RETSINA* entretanto, previne que tais problemas aconteçam.

- **Mapeamento de nome para localização:** Este componente provê um significado à localização física do agente através do mapeamento em um ID. Assim, quando um agente precisa saber o endereço de um outro agente, ele pergunta para o componente de mapeamento.

2.4.2 STEAM Framework

O *STEAM*(TAMBE, 1997a) é um modelo geral de *teamwork* construído com base nos trabalhos teóricos de *Joint Intentions*(LEVESQUE; COHEN; NUNES, 1990) e de *SharedPlans*. Ele foi desenvolvido com o objetivo de contornar problemas envolvidos no desenvolvimento de times de agentes para ambientes complexos. Isto porque, até então os sistemas multiagentes implementados freqüentemente falhavam em fornecer a flexibilidade necessária na coordenação e comunicação para manter a coerência do time em tais domínios. A maioria desses sistemas era suprida com planos pré-definidos e coordenação específica para o domínio. Quando colocados em um domínio dinâmico com incertezas, a inflexibilidade da coordenação pré-planejada leva a grandes falhas, pois em muitos casos é praticamente impossível pré-planejar todas as situações possíveis. Com o crescimento da complexidade das situações de *teamwork*, essas abordagens aumentavam os casos de falha. Outro problema existente era o fato da coordenação estar ligada a domínios específicos, o que inviabiliza o reuso em outros domínios.

Assim, o *STEAM* surge para prover aos agentes um modelo geral de *teamwork* que possibilita contornar essas dificuldades. Tal modelo permite ao agente raciocinar de forma autônoma sobre a coordenação e a comunicação, provendo os requisitos de flexibilidade no *teamwork*. Este modelo genérico também permite o reuso da capacidade de *teamwork* em diversos domínios. Tal reuso não apenas economiza tempo de implementação, como também garante a consistência do *teamwork* através das aplicações.

No *STEAM*, *Joint Intentions* são usadas como blocos de construção do *teamwork*. Algumas vantagens são tiradas disso. A primeira é que o comprometimento em *Joint Intention* provê o princípio para o raciocínio sobre a coordenação e a comunicação em *teamwork*. Segunda, a junção de comprometimentos em *Joint Intentions* provê o guia para monitorar e manter as atividades do time. Terceiro, um *Joint Intention* leva para uma representação explícita de uma atividade de time, e assim facilita o raciocínio sobre *teamwork*.

Adicionalmente, para garantir a coerência do time, o *STEAM* utiliza o *SharedPlans* para resolver questões como as seguintes:

- Evitar que um agente interfira de forma destrutiva nas ações de outros.
- Controlar a quantidade de informação compartilhada.
- Replanejar quando o objetivo é visto como inalcançável.

- Generalizar a capacidade de comunicação.

Membros do time devem possuir um caminho de solução comum em seus *Joint Intentions* para um objetivo de alto-nível. Sem essa restrição, membros do time poderiam possuir soluções alternativas que cancelam uns aos outros, não gerando progresso em direção ao objetivo. A teoria de *SharedPlans* resolve tal coerência seguindo o *intentions-that* voltado ao objetivo. A abordagem do *STEAM* é paralela com *SharedPlans*, entretanto ele é construído sobre *Joint Intentions*. O *STEAM* usa *Joint Intentions* como bloco de construção para hierarquicamente construir um estado mental de atitudes do membro do time individual e garantir que os membros do time possuam um caminho de solução comum. Em domínios dinâmicos, dado planos reativos, uma receita R_α pode evoluir passo a passo durante a execução. No *STEAM*, como a receita evolui, se o passo β_i requer execução por todo o time, o *STEAM* requisita que todo o time concorde com β_i e forme um *Joint Intention* para executá-lo. Para executar um sub-passo de β_i outros *Joint Intentions* são formados, levando a uma hierarquia. Durante a expansão da hierarquia, se um passo envolve apenas um sub-time, então este sub-time deve formar um *Joint Intentions* para executar esse passo. Se só um indivíduo está envolvido no passo, este forma uma intenção de fazer o passo. Em geral, o resultado da hierarquia de intenções evolui dinamicamente, dependendo da situação em que o time se encontra.

SharedPlans também ajuda a resolver questões como a quantidade de informação que os membros do time devem manter sobre os outros, particularmente quando o passo β_i envolve apenas um sub-time ou um indivíduo. O *STEAM* requer que caso o passo β_i seja executado por um sub-time, membros do time mantenham um rastro das *Joint Intentions* do sub-time para executar o passo. Este rastro intencional não precisa envolver detalhes do plano. Um membro do time deve apenas ser apto a inferir o que os outros membros pretendem fazer no passo β_i.

O *STEAM* precisa formar um *Joint Intention* para replanejar sempre que um objetivo de time para executar β_i é visto como inalcançável. O replanejamento pode levar o time a analisar a causa da não alcançabilidade. Dentre outras possibilidades, a causa poderia ser a ausência de uma sub-tarefa para o sub-time ou indivíduo, ou a falha de um sub-time ou indivíduo na execução de uma tarefa relevante. Em tal caso, cada membro do time atua para determinar o agente apropriado ou o sub-time para executar a tarefa relevante. Como resultado, um agente pode se voluntariar, ou sugerir outro indivíduo ou sub-time para executar a tarefa.

A capacidade de comunicação do *STEAM* é generalizada via abordagem híbrida que combina *Joint Intention* como aspectos de *SharedPlans*. A comunicação de *Joint Intention* poderia potencialmente ser alcançada em *SharedPlans* através de axiomas definidos como *intention-that*. Como por exemplo, considerando que um membro do time obteve informações privadas sobre o sucesso da ação β_1 do time. Em *Joint Intentions* esse membro do time tentará uma crença mútua de que β_1 teve sucesso, levando à comunicação. Em contraste, em *SharedPlans*, a comunicação de membros do time deveria aparecer porque uma ação β_2 segue β_1, e o time não pode fazer β_2 sem todos tomarem conhecimento que β_1 foi feita com sucesso. Assim, baseado na interrelação entre as ações, é possível raciocinar e derivar a comunicação em *SharedPlans*, mas *Joint Intentions* provê tal comunicação sem a necessidade de raciocínio.

Desta forma, o *STEAM* inicia com *Joint Intentions*, mas quando constroi a estrutura hierárquica ele tem um paralelo com a teoria de *SharedPlans*, em particular, *Partial Shared-*

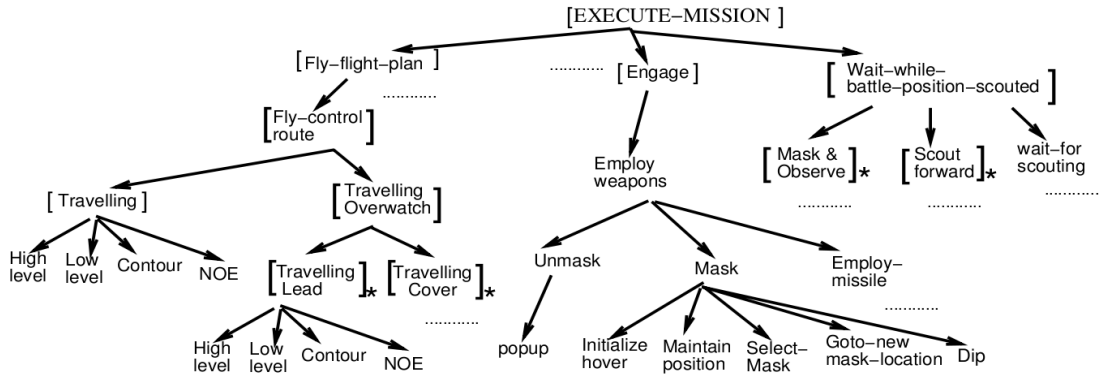

Figura 2.3: Hierarquia de operadores para o domínio de ataque aéreo

Plans. O resultado pode ser considerado como modelo híbrido, que une a formalização do comprometimento de *Joint Intentions* e o tratamento detalhado das atitudes do time em tarefas complexas do *SharedPlans*.

A base do *STEAM* é a execução hierárquica de planos reativos. Estes planos reativos são implementados com operadores em *SOAR*(LEWIS, 1999). O *STEAM* introduz a idéia de operador de time que, quando instanciado por um agente, forma uma *Joint Intention* do time. Os operadores de time expressam explicitamente uma junção de atividades de time, diferente do operador individual regular que expressa a atividade do próprio agente. Na hierarquia apresentada na figura 2.3, os operadores de time são envolvidos em colchetes, enquanto os outros são operadores individuais.

Como os operadores individuais, os operadores de time também consistem de regras de precondições; regras de aplicação; e regras de terminação. Dado um operador *OP* arbitrário, todos os membros do time devem simultaneamente selecionar *OP* para estabelecer um *Joint Intention*. Na Figura 2.3, em alto nível, o time forma um *Joint Intention* para $[execute - mission]_\theta$, na execução desse *Joint Intention*, o time pode fomar o `Joint Intention` $[engage]_\theta$ e para executar $[engage]_\theta$, todos os membros do time devem selecionar individualmente os operadores $employ - weapongs$ formando intenções individuais. Assim, uma hierarquia inteira de intenções individuais e conjuntas é formada quando um agente participa de um trabalho em grupo.

Apesar de representar um marco no desenvolvimento de *teamwork*, a utilização do *STEAM* em sistemas atuais é dificultada devido a existência de complicadores. Um deles é o sistema baseado em regras (*SOAR*) sobre o qual o *STEAM* foi desenvolvido. Como o *STEAM* é representado por um conjunto de operadores e regras genéricas de time na linguagem do *SOAR 7*, mudanças na linguagem utilizada por versões mais recentes do *SOAR* impõe a necessidade reescrever algumas das regras e operadores do *STEAM*, o que pode inviabilizar a sua utilização. O próprio grupo desenvolvedor do *STEAM* fala do *Machinetta* como uma evolução do *STEAM*.

2.4.3 Machinetta Proxy

O *Machinetta* é uma ferramenta para o desenvolvimento de time de agentes que derivou de trabalhos como o *STEAM* e o *TEAMCORE* (TAMBE et al., 1999). Ele usa uma

abordagem de *proxy*, ou seja, faz um papel de intermediador de time para agentes heterogêneos, podendo formar inclusive time contendo agentes artificiais, humanos e robôs. Uma das suas principais inovações frente aos seus antecessores é a autonomia ajustável, que permite, por exemplo, que humanos assumam papéis que estejam sendo executados por agentes. O *Machinetta* também introduz um novo algoritmo de alocação de papéis, permitindo, em teoria, a formação de times com grande quantidade de agentes.

Um padrão que tem emergido para a criação de times robustos e altamente heterogêneos é a arquitetura que utiliza *proxies* semi-autonômos para criar um camada homogênea de coordenação para agente altamente heterogêneos (TAMBE et al., 1999). Provendo cada agente com um *proxy* que tem o conhecimento de time, a habilidade de coordenação de cada par agente-*proxy* é uniformizada.

O *proxy* gerencia a coordenação e executa operações de rotina que são necessárias para a cooperação, como: informar aos demais quando um plano já foi completado, não pode mais ser completado ou tornou-se irrelevante; achar RAPs (*Robot-Agent-Person*) para preencher um determinado papel; quando necessário, ajustar o plano que o grupo está seguindo; e compartilhar informação para garantir a operação continua do time, enquanto deixa os RAPs livres para suas atividades. Nessa abordagem, a coordenação do time é alcançada para cada domínio, dando ao membro do time um *proxy* que assume as rotinas de coordenação em nome do RAP membro do time (PYNADATH; TAMBE, 2003).

Assim, um time de *proxies* implementa um Programa Orientado a Time (POT). Um POT é uma descrição das atividades que precisam ser executadas em nível de time para alcançar o objetivo. Ele consiste de planos de time reativos, papéis, relacionamento entre papéis, e condições para iniciar e terminar um plano. Os *proxies* instanciam os planos dinamicamente quando, durante a execução, o seu estado atual casa com as condições requeridas para lançar o plano. O controle de comunicação do *proxy* determina quais mensagens devem ser enviadas dentre os proxies para garantir que seja mantida a coesão.

Com *proxies* homogêneos é possível escrever um POT que será executado de acordo com o algoritmo de coordenação utilizado pelo *proxy*. Na criação dos POTs os programadores não precisam se preocupar com detalhes específicos de baixo nível de coordenação. Isto porque eles especificam as atividades do time utilizando apenas primitivas de alto nível. Escrevendo POTs neste nível de abstração fica mais fácil para o programador especificar rapidamente atividades complexas de time.

Os planos de time provêem uma representação explícita dos objetivos do time que são mantidos conjuntamente por seus membros. Eles permitem que membros do time simplifiquem seus raciocínios e concentrem-se apenas em suas tarefas que são diretamente relevantes para o atual objetivo do grupo. Assim, eles dão forma ao comportamento do time, usando comportamentos individuais como meios para alcançar um objetivo comum.

Toda a coordenação no *Machinetta* é feita através da alocação de papéis. Os planos de alto nível definidos em um POT determinam quais são as possíveis ações que um time pode executar. Essas tarefas são representadas através de papéis que são atribuídos quando um novo plano é lançado.

Uma contribuição importante do *Machinetta* é seu algoritmo de alocação de papéis, o *LA-DCOP*. Esse algoritmo utiliza uma abordagem de *token*, no qual um *token*, que representa um papel a ser alocado, é passado entre os RAPs até que algum deles aceite ficar com o papel. Assim, a decisão de quem fica com o papel é feita através da verificação de um

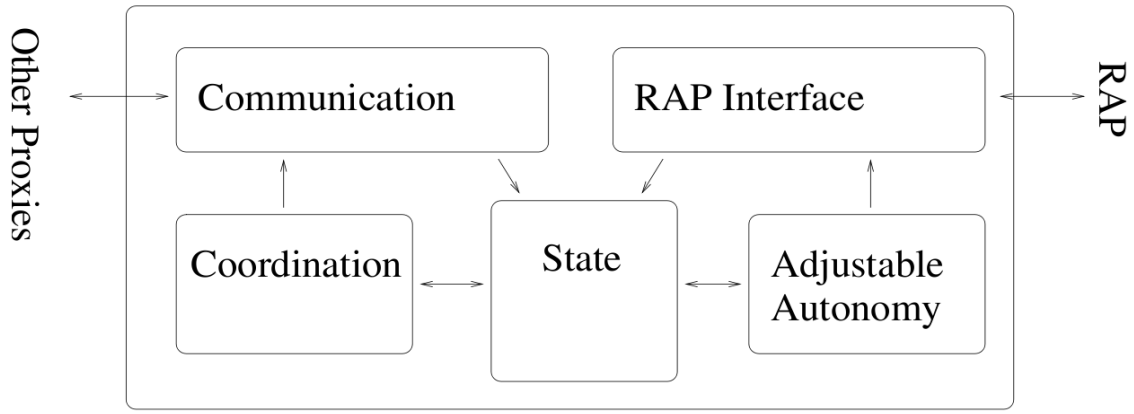

Figura 2.4: Arquitetura do *Machinetta Proxy*.

limiar que o *token* carrega, onde aquele que possui habilidade superior àquela especificada no limiar pré-estabelecido do *token*, fica com o papel.

O *software* do *Machinetta proxy* é formado por cinco componentes, como visto na Figura 2.4.

- Comunicação: trata a comunicação com outros *proxies*.
- Coordinação: trata o raciocínio sobre planos de time e comunicação.
- Estado: a memória de trabalho do *proxy*.
- Autonomia Ajustável: raciocina sobre quando atuar de forma autônoma ou passar o controle para outro membro do time.
- Interface RAP: trata a comunicação com os membros do time.

Os *proxies* por si só não possuem a habilidade de alcançar objetivos no nível do domínio. Ao invés disso, eles precisam garantir que todas as necessidades requisitadas em nível de domínio são atendidas através da instanciação de papéis apropriados atribuídos aos RAPs devidos.

Toda a execução do *proxy* é dirigida à mensagem. Quando um mensagem chega de um RAP ou de outro *proxy*, uma nova crença é adicionada ao estado do *proxy*. As crenças no estado constituem o conhecimento do *proxy* sobre a situação do time e do ambiente. Este estado opera como um quadro-negro, com componentes escrevendo informações e outros reagindo a informação escritas.

Para o *Machinetta*, criar o entendimento compartilhado entre humanos e agentes membros de um time é o maior desafio face ao desenvolvimento de iniciativas mistas com organizações de humanos e agentes. O componente de autonomia ajustável resolve as circunstâncias sobre quando o *proxy* deveria agir de forma autônoma ou quando deveria esperar por entrada de um membro do time. Tal raciocínio é vital para o sucesso da implantação de times heterogêneos contendo pessoas. Entretanto, os outros componentes do *proxy* são isolados do processo de raciocínio pelo componente de autonomia ajustável, assim esses componentes precisam conhecer apenas se a última decisão feita pelo *proxy* foi realizada de forma autônoma ou por um membro do time.

Através da autonomia ajustável, um agente pode dinamicamente variar o nível de autonomia com o qual ele atua, permitindo explorar habilidades humanas para melhorar sua performance de tarefas, sem tornar-se necessariamente dependente de uma interação humana. Por este motivo, a autonomia ajustável é vista como um ponto crítico para o sucesso de um sistema multiagente com suporte à interação humana.

A formação de times com humanos e agentes artificiais carrega alguns desafios que vão além da interação individual entre humano e agente artificial. O time pode ter sua coordenação afetada enquanto espera uma resposta humana, um time de agentes artificiais pode potencialmente tornar globais decisões que são inaceitáveis para times de humanos, e a diversidade da organização humana no mundo real pode necessitar que agentes gradualmente aprendam modelos individualizados dos humanos membros do time, tomando decisões razoáveis mesmo antes de ter informações suficientes disponíveis.

No *Machinetta*, o componente de interface RAP é o principal ponto de extensão do *proxy* para integração com um novo RAP. Nesta interface é tratada a forma como o RAP e o *proxy* compartilham informação. Por exemplo, a interface RAP para um bombeiro que está atendendo uma situação de desastre poderia ser uma grande interface gráfica, enquanto para um agente artificial bastaria uma comunicação simples via *socket*.

3 TWPROXY E O MODELO DE TEAMWORK PROPOSTO

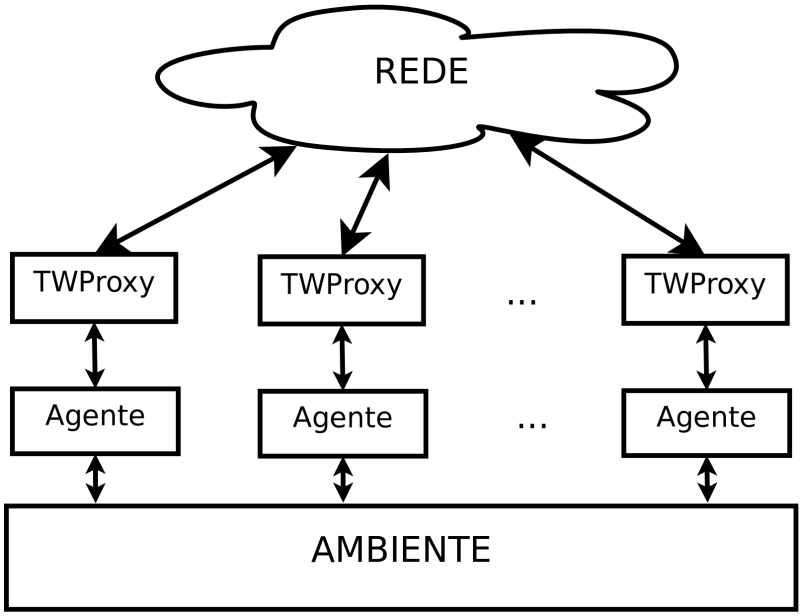

Figura 3.1: Modelo de *teamwork* baseado em *proxy*.

Este capítulo apresenta o modelo utilizado pelo *TWProxy*, uma nova infraestrutura leve e eficiente para prover coordenação para times de agentes heterogêneos, com a finalidade de atender aos requerimentos para o desenvolvimento de times de agentes em ambientes dinâmicos de tempo-real. *TWProxy* é baseado no formalismo de *Joint Intentions* e inspirado no *Machinetta*, adicionando novas características importantes como:

- uma linguagem simples e extensível para descrição de crenças e planos,
- dois processos eficientes de alocação de tarefas parcialmente distribuídos,
- manutenção de consistência através de comunicação atômica,
- reuso de planos terminados,
- planos invariantes ao número de agentes.

Apesar de existirem soluções de *teamwork* para ambientes dinâmicos, não existia até agora uma preocupação evidente com o tempo de resposta no processo de mudança de atividades dentro do time, a ponto de atender requisitos de tempo real. Essa característica é muito importante quando o ambiente é altamente dinâmico e a qualidade das ações

do time pode cair muito com o atraso na mudança de uma estratégia. Por isso, o *TW-Proxy* mantém o foco na diminuição do tempo de resposta, sem perder a qualidade do gerenciamento das ações de time.

Na abordagem apresentada neste trabalho, cada agente é associado com um *proxy*, como visto na Figura 3.1. O *proxy* fornece ao agente uma interface de time, permitindo que ele participe de uma equipe sem se preocupar com os detalhes de coordenação. Tudo que ele precisa para compor um time é agir conforme as sugestões do *proxy* e informar a seu *proxy* sobre a atualização de determinadas crenças. Assim, os *proxies* ficam responsáveis por manter a coerência de time através da comunicação *inter-proxy* e do planejamento realizado pelo grupo de *proxies*. O acoplamento simbiôntico do agente individual com o seu *proxy* é o que forma o agente social. Toda a comunicação e planejamento de time são feitos através dos *proxies*, sem necessidade de consciência do agente individual.

A forma utilizada pelo *TWProxy* para alcançar a coordenação multiagente é através da alocação de papéis, que pode ser vista também como a alocação de tarefas ou atividades de time. Essa maneira de coordenar os agentes é flexível o suficiente para permitir que agentes heterogêneos componham a equipe, já que o *proxy* não diz como o agente deve executar as atividades. O conhecimento de como executar a tarefa fica por conta do próprio agente e este deve ter consciência do que ele está capacitado para fazer. Assim, quando ele informar ao *TWProxy* sobre suas capacidades estará habilitando o *proxy* a realizar uma alocação adequada. Como os *proxies* comunicam-se entre si, cada *proxy* conhece as habilidades dos demais membros da equipe.

O processo de alocação de papéis é disparado por planos, que são compostos por pré-condições, pós-condições e papéis a serem alocados. Quando as pré-condições de um plano são alcançadas, inicia-se o processo de alocação dos papéis especificados pelo plano. Quando uma pós-condição é alcançada, os membros da equipe são liberados dos papéis associados ao plano. Devido a esta coordenação em alto nível, é possível alcançar um *teamwork* flexível, deixando os detalhes de como executar o papel para o agente individual.

O modelo de alocação de papéis é similar a uma organização de trabalho em equipe no mundo real. Quando um grupo de pessoas precisa executar alguma tarefa conjunta, elas normalmente dividem a atividade em subtarefas que cada uma possa executar. Ou seja, no mundo real as pessoas também usam a alocação de papéis para coordenar ações. Assim, a responsabilidade é distribuída, e se existe um comprometimento entre os membros do grupo, todo o time passa a agir de forma consistente.

Como o *TWProxy* é um *proxy* de time baseado em *Joint Intentions*, ele habilita os agentes individuais ao comprometimento com o time, criando uma junção de estados mentais dos membros de um time. Tudo com que o agente individual precisa se preocupar é em manter atualizado o estado de algumas de suas crenças inviduais pré-estabelecidas e em executar os papéis atribuídos. Assim, os *proxies* se comprometem em tornar de conhecimento comum o fato de um objetivo ser concluído, impossível de se atingir ou deixar de ser importante.

O *TWProxy* também provê a funcionalidade de quadro-negro distribuído, pois mantém o conhecimento compartilhado consistente. Este repositório de crenças pode ajudar os agentes a tomarem decisões. Como o ambiente é parcialmente observável e as percepções dos agentes possuem restrições, o conhecimento compartilhado pelo *TWProxy* reflete um

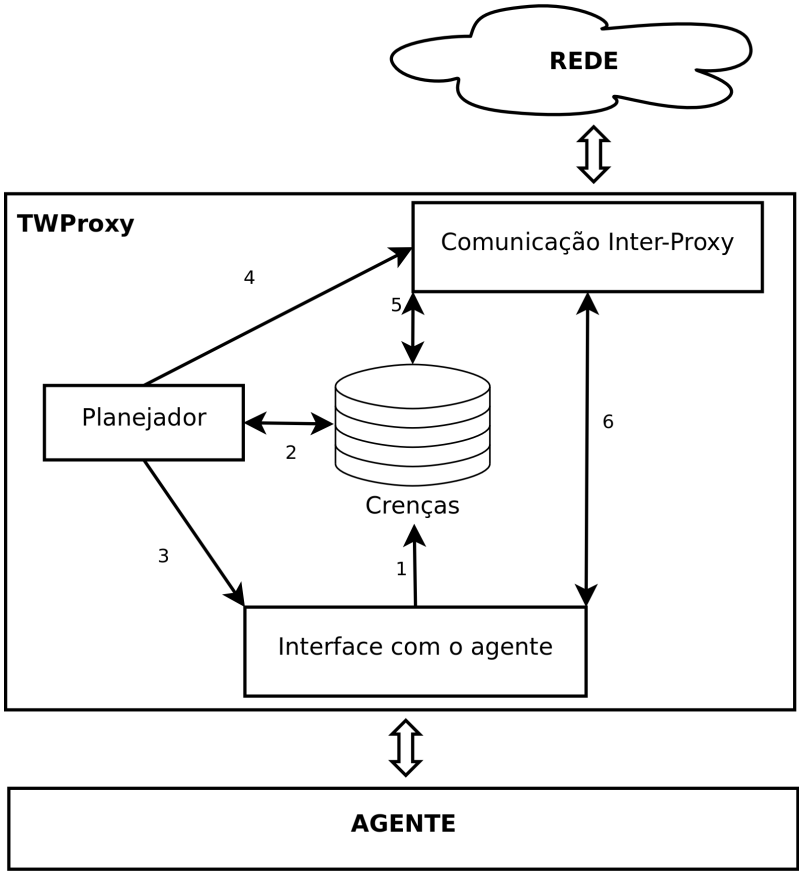

Figura 3.2: *TWProxy* organização.

conjunto de crenças que não poderia ser diretamente percebido por um único agente em específico.

A Figura 3.2 mostra a organização interna do *TWProxy*, composta pelo seu planejador, sua base de conhecimento, um módulo de comunicação com o agente, um módulo de comunicação inter-*proxy* e a interação entre esses elementos. Os módulos de comunição são facilmente extensíveis para atender a novos requisitos de organização, e também para possibilitar a integração do *TWProxy* com novos agentes sem a necessidade de alterar o planejador ou a base de crenças. O planejador e a base de conhecimento são dirigidos pela linguagem de definição de planos e crenças apresentados na seção 3.1. A seguir são descritas as interações internas do *TWProxy* representadas por setas na Figura 3.2:

1. Chegada de uma nova crença ou a atualização de alguma crença existente que foi percebida pelo agente e que irá atualizar a base de conhecimento.

2. Uso da base de conhecimento pelo planejador, que faz uso das crenças armazenadas na base para verificar as condições de ativação e término dos planos, e também para consultar as crenças sobre as capacidades dos membros do time.

3. Deliberação sobre a alocação de um determinado papel para o agente associado ao *proxy* ou envio de requisição para atualização das capacidades.

4. Deliberação sobre a alocação de um determinado papel para um outro membro do time ou envio de requisição para atualização das capacidades, sendo enviado para o *proxy* responsável, que requisitará a seu agente associado.

5. Chegada de nova crença proveniente de outro *proxy*. Também indica o envio de uma crença atualizada para os outros *proxies*.
6. Canal que permite a comunicação entre agentes utilizando o *TWProxy* apenas como um meio. Este canal não é utilizado pelo modelo de coordenação do *TWProxy* e está disponibilizado apenas para atender a requisitos de domínios específicos, onde os agentes precisem realizar algum tipo de comunicação extra que não seja representada como crença de time ou crença de domínio.

3.1 Descrição de planos e crenças

No *TWProxy*, a base de crenças armazena igualmente as crenças iniciais e aquelas criadas em tempo de execução. As crenças iniciais são carregadas a partir de um arquivo estruturado que define um conjunto de crenças básicas para o funcionamento do *proxy*. As crenças geradas de tempo de execução são criadas pelos agentes e podem representar, além de novas crenças, atualizações das crenças iniciais. Esta seção trata das crenças iniciais, juntamente com os planos de time, que são descritos por um arquivo estruturado e carregado na inicialização do *proxy*.

3.1.1 Linguagem

As crenças e planos iniciais dos agentes são armazenados em arquivos estruturados sobre a linguagem apresentada na listagem 3.1. Este arquivo contém crenças sobre: as capacidades do agente, a composição do time e o conhecimento do domínio específico da aplicação. Tal arquivo também armazena os planos de time em alto nível, formando um programa de time. Cada crença é composta por um identificador precedido de "belief"e seguido por um conjunto de pares atributo-valor delimitado por chaves. De forma semelhante, cada plano possui um identificador precedido de "plan"e seguido por: um conjunto de papéis a serem alocados, expressões de pré-condições para ativação e expressões de pós-condições para finalização do plano.

Listagem 3.1: Gramática da linguagem que define os planos e as crenças iniciais.

```
          <blf> ::= <block> | <block> <blf>

        <block> ::= "belief" <belief_block> |
                    "plan" <plan_block>

 <belief_block> ::= <id> "{" <inside_belief> "}"

<inside_belief> ::= <attr_value> |
                    <attr_value> <inside_belief>

   <attr_value> ::= <id> ":" <value> ";"

        <value> ::= <id> |
                    <id> "," <values> |
                    <number> |
                    <number> "," <value> |
                    "true" | "false"

   <plan_block> ::= <id> "{" <inside_plan> "}"

  <inside_plan> ::= "roles" ":" <role_values> ";"
                     <pre_block> <post_block>

  <role_values> ::= <id> |
```

```
                    <id> "," <role_values> |
                    <id> "+" |
                    <id> "+" <role_values>

    <pre_block> ::= "precondition" "{" <expr> "}"

   <post_block> ::= "postcondition" "{" <expr> "}"

         <expr> ::= "(" <expr> ")" |
                     <id> "." <id> <comp> <number> |
                     <id> "." <id> <comp> <id> "." <id> |
                     <id> "." <id> <comp> "true" |
                     <id> "." <id> <comp> "false" |
                     <expr> "|" <expr> |
                     <expr> "&" <expr>

           <id> ::= "[a-zA-Z][a-zA-Z_0-9]*"

       <number> ::= "[+-]?[0-9]+|[+-]?[0-9]+\.[0-9]+"

         <comp> ::= "==" | "!=" | "<"  | "<=" | ">"  | ">="
```

Do ponto de vista conceitual, o par agente e *proxy* executa um programa de time conhecido como TOP (Team-Oriented Program (SCERRI et al., 2003c)). Esse TOP especifica os planos que devem ser executados conjuntamente pelo time e a interdependência entre esses planos, formando um encadeamento de atribuições de papéis. Tal TOP especifica as atividades em alto nível, sem detalhamentos necessários para a coordenação e comunicação.

Listagem 3.2: Um exemplo de crença sobre capacidade do agente.

```
belief capability_agent001_CTFProtectTheBase {
        type: capability;
        rapId: agent001;
        roleId: CTFProtectTheBase;
        ability: 76;
}
```

As listagens 3.2,3.3,3.4 e 3.5 apresentam exemplos de diferentes partes do arquivo de descrição de planos e crenças para o domínio de *Capturar a Bandeira* (uma modalidade clássica em jogos de primeira pessoa). A listagem 3.2, por exemplo, define que o agente *agent001* pode executar o papel *CTFProtectTheBase* e seu nível de habilidade para este papel é 76. O identificador *capability_agent001_CTFProtectTheBase* é utilizado para a recuperação da crença na base, que utiliza uma estrutura de dicionário para acesso eficiente.

Listagem 3.3: Um exemplo de crença sobre a composição do time.

```
belief teamComposition {
    type: teamComposition;
    members: agent001, agent002, agent003;
    myself: agent001;
    agent001_host: localhost;
    agent002_host: localhost;
    agent003_host: localhost;
    agent001_port1: 6001;
    agent002_port1: 6002;
    agent003_port1: 6003;
    agent001_port2: 7001;
    agent002_port2: 7002;
    agent003_port2: 7003;
}
```

A listagem 3.3 mostra a crença sobre a composição do time, onde os membros são descritos através de uma lista de valor. Nessa mesma crença de composição do time existe o

identificador do *proxy*, que nesse caso é o *agent001*. Os demais atributos-valor definem como devem ser estabelecidas as conexões entre os *proxies*.

Listagem 3.4: Um exemplo de crença de domínio específico.

```
belief flag_enemy{
        type: flag;
        owner: enemy;
        have: false;
}

belief flag_friend{
        type: flag;
        owner: friend;
        have: true;
}
```

As crenças de domínio específico também são descritas no arquivo planos e crenças, como é mostrado na listagem 3.4. O conteúdo da crença é flexível para descrever novas propriedades, sendo possível adicionar um novo atributo dinamicamente. Ou seja, contanto que seja respeitada a gramática definida na listagem 3.1, novas crenças podem ser modeladas através de adição de novos pares atributo-valor.

Listagem 3.5: Um exemplo de plano

```
plan p1{
        roles: CTFProtectTheBase, CTFCaptureTheFlag+;

        precondition{
                ( flag_friend.have == true ) &
                ( flag_enemy.have == false )
        }

        postcondition {
                ( flag_friend.have == false ) |
                ( flag_enemy.have == true )
        }
}
```

A listagem 3.5 apresenta um exemplo de plano para o contexto de *Capturar a Bandeira*. Quando um time tem a sua bandeira e não possui a bandeira do inimigo, o plano *p1* é ativado, lançando o processo de alocação de papéis. Este plano especifica dois papéis a serem alocados, o *CTFProtectTheBase* e o *CTFCaptureTheFlag*, onde o segundo é flexível para mais de uma alocação, o que é indicado pelo símbolo "+"no final do nome do papel. A utilização de papéis flexíveis permite que um plano seja invariante ao número de agentes, ou seja, mesmo mudando a quantidade de agentes, é possível utilizar o mesmo plano a depender de como sejam especificados os papéis. No exemplo da listagem 3.5 a sentença *"roles: CTFProtectTheBase, CTFCaptureTheFlag+;"* informa que um agente deve assumir o papel *CTFProtectTheBase* e todos os demais devem assumir o papel *CTFCaptureTheFlag*. É possível especificar o mesmo papel mais de uma vez, por exemplo, *"roles: CTFProtectTheBase, CTFPRotectTheBase, CTFCaptureTheFlag+;"* indica que dois agentes devem executar *CTFProtectTheBase* enquanto os demais devem assumir o papel *CTFCaptureTheFlag*. A utilização de mais de um papel flexível implica na distribuição uniforme desses papéis entre os agentes não alocados. Ou seja, dado um exemplo com *"roles: CTFProtectTheBase, CTFCaptureTheFlag+, CTFRecoverAnyFlag+;"*, um agente é alocado para executar *CTFProtectTheBase* enquanto os demais são uniformemente distribuídos entre *CTFCaptureTheFlag* e *CTFRecoverAnyFlag*.

3.1.2 Arquivo completo de exemplo

A Listagem 3.6 contém o arquivo de crenças e planos utilizados pelo *proxy* do agente *agent001* no experimento de *Capturar a Bandeira*. Cada *proxy* possui seu arquivo de crenças e planos. As crenças e os planos são definidos em arquivo formando o programa do time, onde os planos são utilizados como regras para lançar o processo de alocação de papéis e as crenças são armazenadas para serem utilizadas nos planos.

Listagem 3.6: Arquivo de crenças e planos do Proxy 1.

```
// ==========================
//   Agent 001 _ Capability
// ___________________________

belief capability_agent001_CTFRecoverAnyFlag {
        type: capability;
        rapId: agent001;
        load: 100;
        roleId: CTFRecoverAnyFlag;
        ability: 100;
}

belief capability_agent001_CTFScout {
        type: capability;
        rapId: agent001;
        load: 100;
        roleId: CTFScout;
        ability: 100;
}


belief capability_agent001_CTFRecoverMyFlag {
        type: capability;
        rapId: agent001;
        load: 100;
        roleId: CTFRecoverMyFlag;
        ability: 100;
}

belief capability_agent001_CTFGoBackToTheBase {
        type: capability;
        rapId: agent001;
        load: 100;
        roleId: CTFGoBackToTheBase;
        ability: 100;
}


belief capability_agent001_CTFProtectTheBase {
        type: capability;
        rapId: agent001;
        load: 100;
        roleId: CTFProtectTheBase;
        ability: 100;
}

belief capability_agent001_CTFCaptureTheFlag {
        type: capability;
        rapId: agent001;
        load: 100;
        roleId: CTFCaptureTheFlag;
        ability: 100;
}

// ==========================
//      Team Composition
// ___________________________

belief teamComposition {
        type: teamComposition;
        members: agent001, agent002, agent003, agent004, agent005;
```

```
        myself: agent001;
        agent001_host: 127.0.0.1;
        agent002_host: 127.0.0.1;
        agent003_host: 127.0.0.1;
        agent004_host: 127.0.0.1;
        agent005_host: 127.0.0.1;
        agent001_port1: 6001;
        agent001_port2: 7001;
        agent002_port1: 6002;
        agent002_port2: 7002;
        agent003_port1: 6003;
        agent003_port2: 7003;
        agent004_port1: 6004;
        agent004_port2: 7004;
        agent005_port1: 6005;
        agent005_port2: 7005;
}


// ==========================
//    Initial Flag Belief
// __________________________

belief flag_enemy{
        type: flag;
        owner: enemy;
        have: false;
}

belief flag_friend{
        type: flag;
        owner: friend;
        have: true;
}



// ==========================
//       Plan Definition
// __________________________

// I have my flag, but I do not have the enemy flag
plan p1{
        roles: CTFProtectTheBase, CTFProtectTheBase, CTFCaptureTheFlag+;

        precondition{
                ( flag_friend.have == true ) &
                ( flag_enemy.have == false )
        }

        postcondition {
                ( flag_friend.have == false ) |
                ( flag_enemy.have == true )
        }
}

// I have my flag and I have the enemy flag
plan p2{
        roles: CTFGoBackToTheBase, CTFProtectTheBase, CTFRecoverAnyFlag+;

        precondition{
                ( flag_friend.have == true ) &
                ( flag_enemy.have == true )
        }

        postcondition {
                ( flag_friend.have == false ) |
                ( flag_enemy.have == false )
        }
}

// I do not have my flag, but I have the enemy flag
plan p3{
```

```
        roles: CTFProtectTheBase+, CTFRecoverMyFlag+;

        precondition{
                ( flag_friend.have == false ) &
                ( flag_enemy.have == true )
        }

        postcondition {
                ( flag_friend.have == true ) |
                ( flag_enemy.have == false)
        }
}

// I do not have my flag and I do not have the enemy flag
plan p4{
        roles: CTFProtectTheBase, CTFRecoverAnyFlag+;

        precondition{
                ( flag_friend.have == false ) &
                ( flag_enemy.have == false )
        }

        postcondition {
                ( flag_friend.have == true ) |
                ( flag_enemy.have == true  )
        }
}
```

3.2 Comunicação

O módulo de comunicação inter-*proxy* usa dois tipos de comunicação, a difusão atômica (DEFAGO; SCHIPER; URBAN, 2004) e a comunicação direta (*unicast*). Cada tipo de comunicação é separada em diferentes canais. A difusão atômica é usada para compartilhar crenças do time, mantendo consistente o quadro-negro distribuído; e a comunicação direta é usada no processo de alocação de papéis.

O *TWProxy* implementa uma forma simples de difusão atômica utilizando *token*, mas é suficientemente flexível para suportar outras abordagens. A Figura 3.3 apresenta o diagrama das classes de comunicação inter-*proxy* do *TWProxy*. A classe *Atomicomm* é a responsável pela implementação da difusão atômica e uma nova herança da classe *Comm* pode permitir tanto o suporte a outras abordagens de difusão atômica como adequação a novas organizações.

A difusão atômica é utilizada quando uma nova crença precisa ser compartilhada com a equipe. Em outras palavras, quando um membro do time percebe uma nova situação que é relevante para a equipe, ele compartilha essa informação mantendo o conhecimento do time consistente. Entretanto, os membros do time não precisam compartilhar seus conhecimentos individuais, diminuindo a quantidade de mensagens trocadas.

O canal de *unicast* é usado para executar uma variação do Contract Net Protocol(SMITH, 1981), a fim de realizar a alocação de papéis. Neste processo, o líder utiliza comunicação direta para requisitar aos membros que atualizem as crenças do grupo sobre suas capacidades; uma vez atualizadas essas crenças, ele decide sobre a atribuição de papéis, enviando, via *unicast*, o papel que cada membro deve executar. Neste processo, a quantidade de mensagens diretas trocadas é baixa, sendo a maior parte da comunicação feita no compartilhamento das crenças sobre capacidade através do canal de difusão.

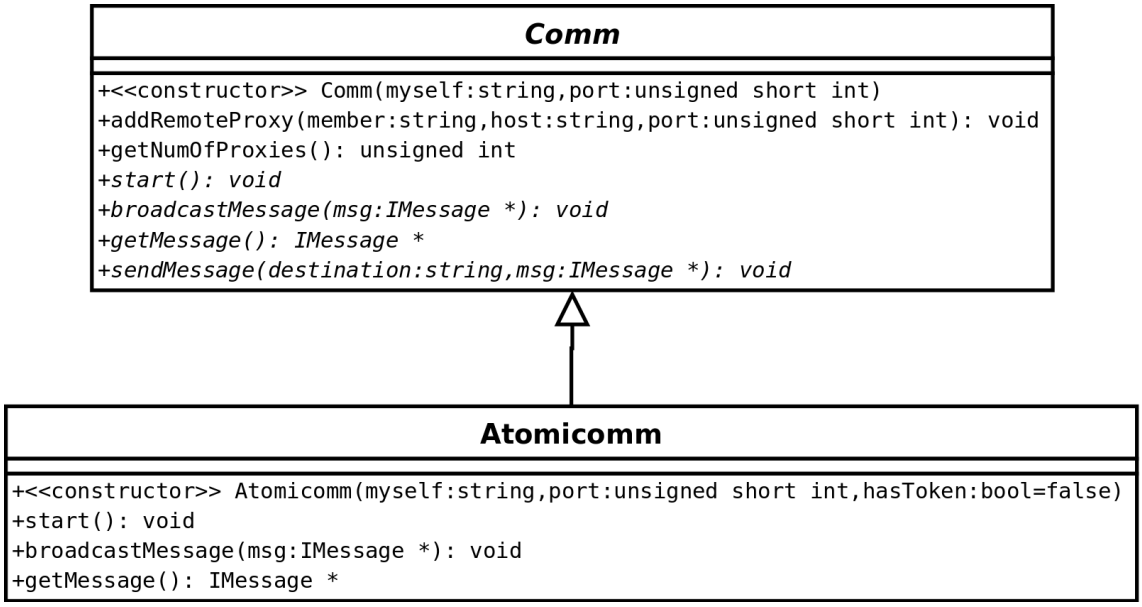

Figura 3.3: Comunicação inter-*proxy*.

3.3 Coordenação

A coordenação sobre o paradigma de *teamwork*, fornecida aos agentes pelo modelo do *TWProxy*, é resultado do processo de alocação de papéis executado pelo conjunto de *proxies*. Este processo é feito de forma parcialmente distribuída, utilizando um líder do time para decidir sobre a alocação. O principal benefício por utilizar um processo de alocação de papéis apenas parcialmente distribuído, centralizando no momento da decisão da alocação, é que existem algoritmos ótimos e quase ótimos com tempo polimonial para resolver a alocação de forma centralizada, possibilitando a economia de tempo no processo de decisão comparando com os processos totalmente distribuídos, apesar de ainda se gastar tempo com comunicação para a execução do processo.

3.3.1 Escolha do líder

O planejador do *TWProxy* utiliza o líder do time para lançar um novo plano. Este líder pode ser prefixado estaticamente ou definido dinamicamente em tempo de execução. O uso de líder de time evita a necessidade de resolver problemas de conflitos na alocação de papéis, economizando tempo para reagir às mudanças do ambiente. O líder do time tem as crenças atualizadas e pode lançar de forma consistente um plano de time, porque todos estão comprometidos a compartilhar informações relevantes para o time.

A escolha do líder pode ser feita de duas formas. A primeira, mais simples, é uma escolha estática, onde um líder é definido no momento da inicialização do *TWProxy*, com base em seu arquivo de configuração. A segunda forma é a escolha dinâmica, onde os membros da equipe decidem quem será o líder em um dado momento.

A escolha dinâmica de um líder baseia-se no pedido de confirmação sobre a liderança. Inicialmente, assume-se que o membro que requisitar liderança primeiro para um processo de alocação de papel será o líder daquele processo. Assim, sempre que um membro da equipe deseja se tornar um líder, pede aos demais para que isso ocorra. Caso não haja

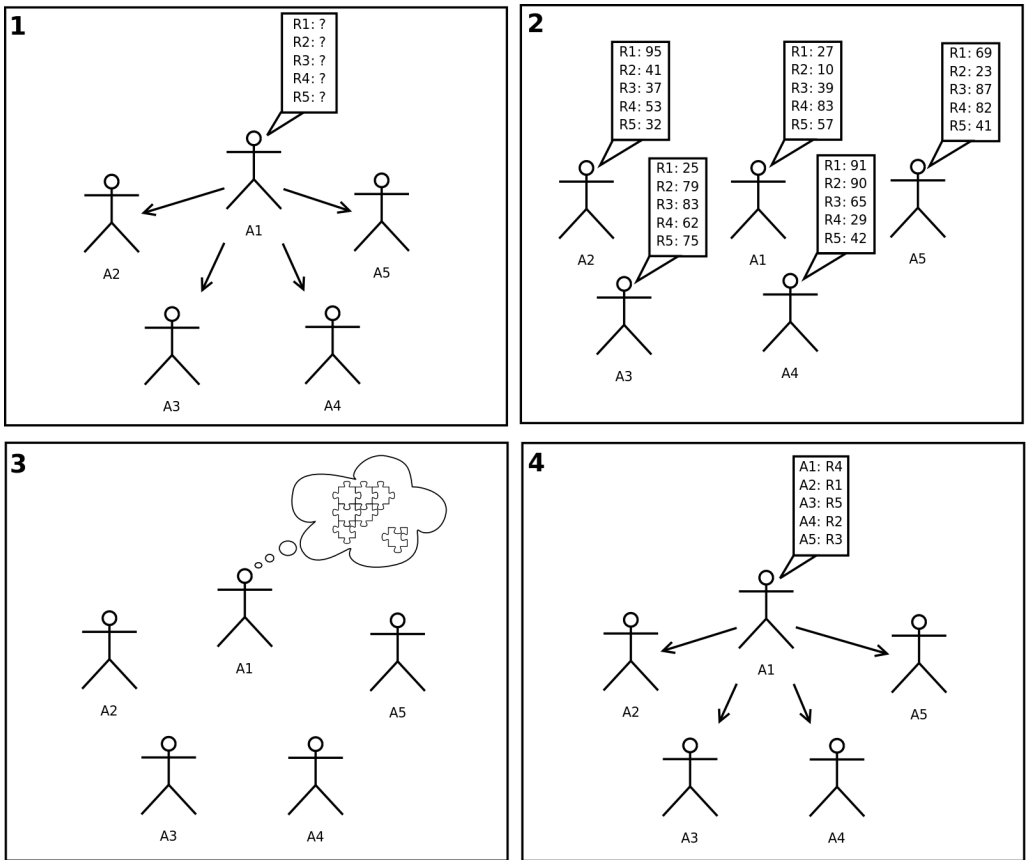

Figura 3.4: Processo de alocação de papéis.

outro candidato, ele torna-se automaticamente o líder. Existindo outro candidato, aquele que pediu a liderança primeiro será o líder.

A decisão sobre quem pediu liderança primeiro fica por conta do próprio canal de comunicação, através do uso da difusão atômica. Dado que as mensagens seguem através do canal de comunicação em ordem total, torna-se trivial identificar qual foi o primeiro pedido de liderança.

A posição de liderança dinâmica é reavaliada sempre que necessária a realocação de papéis, o que pode gerar um certo custo de comunicação. Mesmo não sendo necessário trocar mensagens para resolver conflitos, em ambientes altamente dinâmicos, a escolha de líder de maneira estática pode representar uma maior eficiência, com menor quantidade de mensagens trocadas, apesar de representar uma solução mais centralizada do processo de decisão.

3.3.2 Processo de alocação de papéis

A Figura 3.4 apresenta, em quatro passos, o processo de alocação de papéis utilizado pelo *TWProxy*, considerando que um líder já foi escolhido. No primeiro passo, apresentado no quadro um, o líder (agente A1)requisita aos demais membros do grupo que atualizem as informações sobre suas capacidades no quadro-negro distribuído. No segundo passo, cada membro atualiza as crenças do time sobre suas capacidades por meio de difusão atômica. O líder, que recebe a atualização das crenças sobre as capacidades e habilidades do time,

executa localmente um algoritmo de alocação de tarefas e após a execução do algoritmo ele pode atribuir, no quarto passo, os papéis para os demais membros.

O que poderia deteriorar essa abordagem seria a demora de um membro da equipe em atualizar suas capacidades, deixando o líder esperando de forma ociosa. Mas isso pode ser contornado com a espera com *timeout*, continuando o processo de decisão mesmo que as crenças sobre capacidades não estejam atualizadas. Nestas circunstâncias, a qualidade da alocação dependerá de quando ocorreu a última atualização de crença e o quanto varia a capacidade dos membros com o passar do tempo.

O planejador executa como um sistema baseado em regras. Todos os planos contém pré-condições para ativá-los e pós-condições para pará-los, como visto na listagem 3.5. Essas regras são verificadas usando o conhecimento compartilhado do time. A ativação de um plano dispara o processo de alocação de papéis, como visto na Figura 3.4, e o término de um plano libera os agentes dos papéis previamente alocados.

Com o *TWProxy*, o líder do time pode usar dois algoritmos para executar a alocação: um algoritmo ótimo com complexidade $O(n^3)$ ou um novo algoritmo heurístico com complexidade $O(n^2 log(n^2))$. O algoritmo ótimo é o algoritimo Húngaro ou algoritmo de Kuhn-Munkres, uma solução clássica para o problema de atribuição, originalmente proposto por H. W. Kuhn em 1955 (KUHN, 1955) e refinado por J. Munkres em 1957 (MUNKRES, 1957). O algoritmo heurístico é o *Normalized Relative Weight (NRW)*, proposto neste trabalho e apresentado no quadro de Algoritmo 1.

No problema da alocação de papéis, cada agente, $a_i \in A$, é definido por suas capacidades de executar papéis, sendo $R = r_1, ..., r_k$ o conjunto de papéis. A capacidade de um agente a_i para executar um papel r_j é quantitativamente dada por $C(a_i, r_j)$. A matriz M define a alocação de papéis para os membros do time, onde $m_{i,j}$ é o valor da i-ésima linha e a j-ésima coluna, e $m_{i,j} = 1$ se a_i deverá executar r_j, caso contrário $m_{i,j} = 0$.

Assim, o objetivo desse problema é maximizar a função

$$f(M) = \sum_{a_i \in A} \sum_{r_j \in R} C(a_i, r_j) * m_{i,j}$$

tal que

$$\forall i \sum_{1 \leq j \leq k} m_{i,j} = 1$$

O algoritmo *NRW* é uma modificação de uma estratégia gulosa. Ele considera que cada agente tem um vetor com todas as suas capacidades, e o algoritmo usa apenas um sub-vetor dessas capacidades relacionado aos papéis que devem ser executados. Após a seleção, o *NRW* normaliza o sub-vetor de cada agente. Assim, as capacidades de agentes distintos podem ser comparadas de maneira mais justa. Então, o algoritmo pode usar uma estratégia gulosa para achar uma solução.

O laço da linha 3 realiza a soma dos quadrados das capacidades do agente a_i para executar o papel r_j e na linha 6 é finalizado o cálculo da norma do vetor de capacidades do agente a_i. No laço da linha 7, as capacidades dos agentes a_i são normalizadas, gerando pesos que serão utilizados para ordenação dessas capacidades (linha 13). Nas linhas 9 e 10 são armazenadas informações para identificação do agente e papel a que se refere a capacidade após o processo de ordenação. No laço da linha 14 é feito um percurso sobre o vetor

Algorithm 1 Normalized Relative Weight

Require: C é um sub-conjunto de todas as capacidades e ele contém apenas os papéis que serão atribuídos.

1: **for** $i = 0$ to $nAgents$ **do**
2: $norma \Leftarrow 0$
3: **for** $j = 0$ to $nRoles$ **do**
4: $norma \Leftarrow norma + C(i * nRoles + j).cap^2$
5: **end for**
6: $norma = sqrt(norma)$
7: **for** $j = 1$ to $nRoles$ **do**
8: $C(i * nRoles + j).cap \Leftarrow C(i * nRoles + j).cap/norma$
9: $C(i * nRoles + j).agent \Leftarrow i$
10: $C(i * nRoles + j).role \Leftarrow j$
11: **end for**
12: **end for**
13: $C \Leftarrow sortByCapability(C)$
14: **for** $k = 0$ to $nRoles * nAgents$ **do**
15: $aIdx \Leftarrow C(k).agent$
16: $rIdx \Leftarrow C(k).role$
17: **if** $Allocated(rIdx) =$ **false** and $HasRole(aIdx) =$ **false then**
18: $Allocated(rIdx) \Leftarrow$ **true**
19: $HasRole(aIdx) \Leftarrow$ **true**
20: $Allocation(aIdx) \Leftarrow rIdx$
21: **end if**
22: **end for**

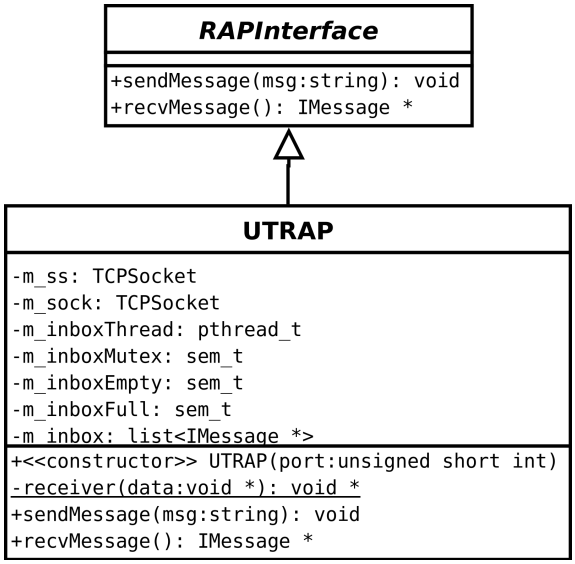

Figura 3.5: Interface com o agente.

ordenado de forma a obter os pares (agente, papel) mais interessantes e que ainda não foram alocados.

3.4 Interface com o agente

Cada agente individual é anexado ao *TWProxy* formando assim um agente social. A comunicação entre o agente individual e o *TWProxy* é definida por uma interface flexível, permitindo interação com vários tipos de agentes.

A interface entre *proxy* e agente é definida por uma classe base (*RAPInterface*, visto na Figura 3.5) que deve ser extendida para implementar essa comunicação. Tal interface é baseada na troca de mensagens e pode ser adaptada para diferentes agentes com a implementação dos métodos *sendMessage* e *recvMessage* através da especialização de *RAPInterface*.

A classe *UTRAP* é um exemplo de interface criada para os agentes utilizados no estudo de caso desse trabalho. Nesta interface, a comunicação é realizada através de conexões via socket. Nela, o método *sendMessage* envia atribuições de papéis e requisições de capacidades, enquanto o método *recvMessage* recebe as crenças que devem ser compartilhadas com o time.

Para a utilização do *TWProxy* em um novo domínio são necessárias as implementações da interface com o agente e da descrição do programa de time. Assim, é possível criar um time para um novo domínio com muito pouco esforço, dado que o agente individual já esteja implementado. O agente não precisa ter um comprometimento social além do compartilhamento de suas crenças, porque o *TWProxy* faz isso por ele. Desta maneira, o desenvolvimento do agente também fica bastante simplificado.

4 EXPERIMENTOS E AVALIAÇÕES

Neste capítulo são apresentados três tipos de experimentos que visam avaliar a ferramenta proposta. O primeiro tipo de experimento avalia o processo de alocação de papéis, comparando o algoritmo húngaro com o heurístico proposto nesta dissertação. O segundo avalia o período de inconsistência de time, medindo o tempo passado entre uma mudança no ambiente e a adoção de uma nova estratégia para o time. O terceiro tipo de experimento avalia a eficiência de ferramentas de *teamwork* em um ambiente de tempo-real onde a organização de time é necessária.

4.1 Eficiência da alocação de papéis

O primeiro experimento sobre a eficiência da alocação de papéis tem o objetivo de explicar o porquê e quando usar o algoritmo heurístico, dado que existe uma solução ótima que executa em tempo polinomial. A primeira resposta é simples, o heurístico escala melhor que o ótimo. Considerando um tempo máximo de 1 segundo para a execução do algoritmo de alocação de papéis, o algoritmo ótimo escala até cerca de 230 agentes, enquanto o heurístico escala até 1200 agentes. Por essa razão, o *TWProxy* usa por padrão o algoritmo húngaro quando o número de agentes é menor que um valor especificado, e o método heurístico é utilizado no caso contrário.

Uma característica importante quando se trata de algoritmo heurístico é conhecer o quão boa é a heurística, e para essa avaliação uma análise estatística soa como um bom caminho. Assim, foram simuladas muitas alocações de papéis e calculado o quão próximo do ótimo a heurística chegou. O resultado de 999 alocações, para uma quantidade de agentes variando de 3 a 100, é apresentado nas Figuras 4.1 e 4.2. A Figura 4.1 representa a média do percentual obtido em relação ao ótimo. Aumentando o número de agentes, o algoritmo tende a 98% do ótimo. A Figura 4.2 mostra como decresce o desvio padrão com o aumento do número de agentes.

4.2 Avaliação do período de inconsistência de time

Uma questão importante nos problemas de *teamwork*, situados em ambientes de tempo-real, é a diminuição do período de inconsistência de time, pois tal tempo tem implicações diretas na eficiência do grupo. Dentro de um domínio de *Capturar a Bandeira*, uma inconsistência de time poderia ser uma situação como a seguinte: dado um time composto

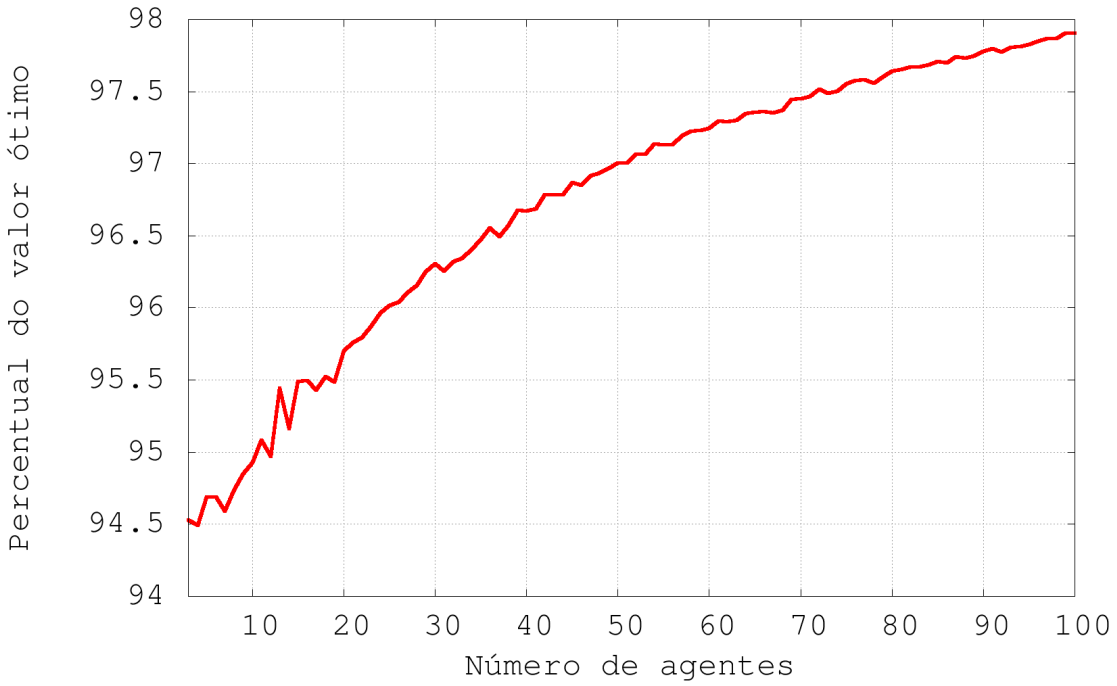

Figura 4.1: Média da percentagem do valor máximo alcançada pela heurística *NRW*.

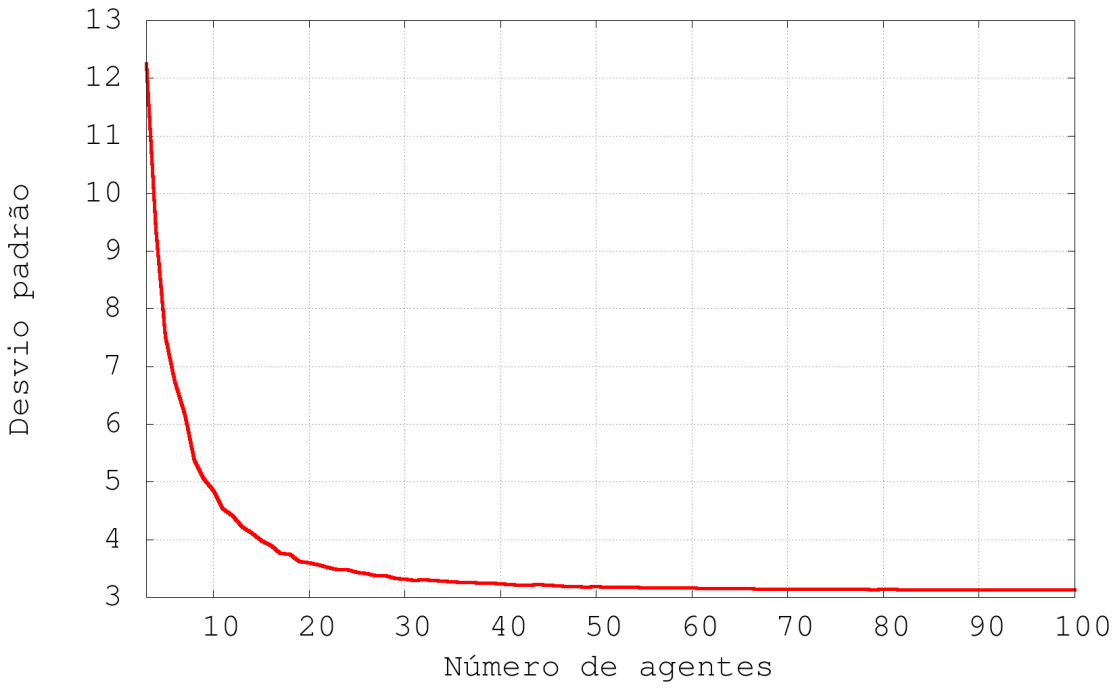

Figura 4.2: Desvio do valor ótimo pela heurística *NRW*.

Tabela 4.1: Média e desvio padrão do Período de Inconsistência de Time.

	Tempo transcorrido(ms)	
	μ	σ
TWProxy	77,91	12,95
Machinetta	12303,34	6459,54

pelos agentes A, B e C, se A ou B estão tentando recuperar uma bandeira que já foi recuperada por C, então o time está em um estado inconsistente, pois o objetivo já foi alcançado por C e mesmo assim A ou B ainda continuam tentando alcançar um objetivo que já foi atingido. Todo time em um ambiente dinâmico permanece em um estado inconsistente durante algum período de tempo, o objetivo é minimizar esse período de inconsistência.

O experimento sobre o período de inconsistência de time tem o objetivo de avaliar a habilidade do *TWProxy* de reagir em tempo-real às mudanças do ambiente. A fim de comparar essa performance, o *Machinetta* foi exposto à mesma situação. Um grupo de agentes enviava atualizações de crenças e o tempo de reação das ferramentas era medido, isto é, o período entre a informação da mudança no ambiente e a adoção de uma nova estratégia pelo time. Este tempo transcorrido entre uma mudança no ambiente e a finalização do processo de alocação de papéis é definido aqui como período de inconsistência de time.

Tanto o *TWProxy* como o *Machinetta* foram expostos à mesma seqüência de mudanças de estado do ambiente. Esta seqüência foi composta por 25 atualizações de ambiente com relevância para o time, representando uma simulação do ambiente de *Capturar a Bandeira*. Cada *proxy* executou 100 vezes essa sequência com a finalidade de obter um comportamento geral do período de inconsistência de time. A latência de rede não está sendo considerada neste experimento.

O resultado apresentado na Tabela 4.1 descreve a média (μ) e o desvio padrão (σ) do conjunto de execuções. A unidade de medida é milisegundos e o tempo representa o período transcorrido entre uma nova crença ser recebida e a modificação da estratégia. Nos experimentos o *TWProxy* alcançou a média de $77.91ms$, enquanto o *Machinetta* obteve $12303.34ms$. Em outras palavras, o *TWProxy* foi, em média, mais de cem vezes mais rápido do que o *Machinetta* para alcançar o estado consistente. O desvio padrão também expressa o quão estável foi a execução do *TWProxy* comparada com a do *Machinetta*.

As vantagens em performance e estabilidade do *TWProxy* contra o *Machinetta* foi o principal resultado deste experimento. Entretanto, outro ponto que merece atenção é o uso de memória do *Machinetta*, com mais de 100MB por *proxy*, tornando-o inviável para o uso em algumas aplicações, como jogos de computador. O *TWProxy*, nesse experimento, não usou mais que 1.6MB de memória. As justificativas para o excesso de memória podem remeter ao tipo de aplicação para o qual o *Machinetta* foi desenvolvido. Enquanto o *TWProxy* foi projetado para atender a requisitos de tempo-real, o *Machinetta* foi originalmente desenvolvido para permitir, de uma forma geral, a participação de humanos em times compostos por robôs e agentes artificiais.

Outros pontos que podem justificar uma diferença tão grande de performance entre o *TWProxy* e o *Machinetta* são: as diferenças dos algoritmos de alocação de papéis, a otimização do acesso às crenças no *TWProxy*, a necessidade para resolver conflitos de alocação no *Machinetta*, a utilização de líder de time no *TWProxy*, e até mesmo a linguagem de

programação utilizada (C++ no *TWProxy* em vez de Java utilizada no *Machinetta*).

4.3 Eficiência de teamwork

Este terceiro tipo de experimento avalia a atuação e a eficiência dos times de agentes em situação real de aplicação, utilizando o *Unreal Tournament 2004*, um jogo multijogador de primeira pessoa (*First-Person Shooter*). O modo clássico de *Capturar a Bandeira* foi escolhido para avaliar a coerência do time, por ser fácil de avaliar os resultados e simples de identificar o comportamento do time. O projeto *GameBOTs* (KAMINKA et al., 2002) foi utilizado para permitir a comunicação dos *bots* com o jogo.

Na modalidade de jogo *Capturar a Bandeira*, um time deve obter pontuação capturando a bandeira inimiga na base do oponente e levando-a para a base aliada. Se o personagem que carrega a bandeira é morto, a bandeira cai ao chão e qualquer um pode pegá-la. Se a bandeira aliada é roubada, o time deve resgatá-la antes que a equipe adversária obtenha pontuação com a bandeira capturada.

O jogo UT2004 fornece o ambiente para os agentes na modalidade de jogo em time de *Capturar a Bandeira*. Através do jogo é possível acompanhar e avaliar as ações do time, pois o próprio UT2004 fornece a opção para se assistir às batalhas como espectador. Devido a este recurso, torna-se simples identificar quais papéis os agentes estão executando e também mensurar a eficiência do time através da pontuação do jogo. O UT2004 provê um ambiente com as seguintes características:

- **parcialmente observável** - o agente, usando sua percepção, não pode acessar todos os estados relevantes do ambiente;
- **estocástico** - o próximo estado do ambiente não é completamente determinado pelo estado atual e pelas ações selecionadas pelos agentes;
- **não episódico** - a qualidade das ações dos agentes depende de suas ações anteriores;
- **dinâmico** - o estado do ambiente pode mudar entre a percepção e a ação do agente;
- **contínuo** - o número de estados do ambiente é ilimitado;
- **multiagente** - existem interações de cooperação e competição entre os agentes.

O projeto *GameBOTs* funciona como uma extensão do controle do jogo, permitindo que personagens sejam controlados via socket TCP. Quando o *bot* está conectado ao jogo, informações sensoriais são recebidas através da conexão com o socket e o agente pode atuar enviando comandos para o jogo. Estes comandos definem o comportamento do personagem dentro do mundo virtual, controlando movimentação, ações de ataque e comunicação com outros personagens. A interação entre um agente e o UT2004 é feita através de troca de mensagens. As mensagens de percepção podem ser síncronas ou assíncronas e sua descrição completa pode ser vista no Apêndice B.

Os agentes implementados para os experimentos foram desenvolvidos com o auxílio do *framework IAF* (MONTEIRO; SANTOS, 2007), utilizando sua arquitetura de agentes e reutilizando os comportamentos reativos existentes. Assim, toda a comunicação com o jogo e os comportamentos elementares, como *MoveTo, OffensiveMoveTo, AttackEnemy e*

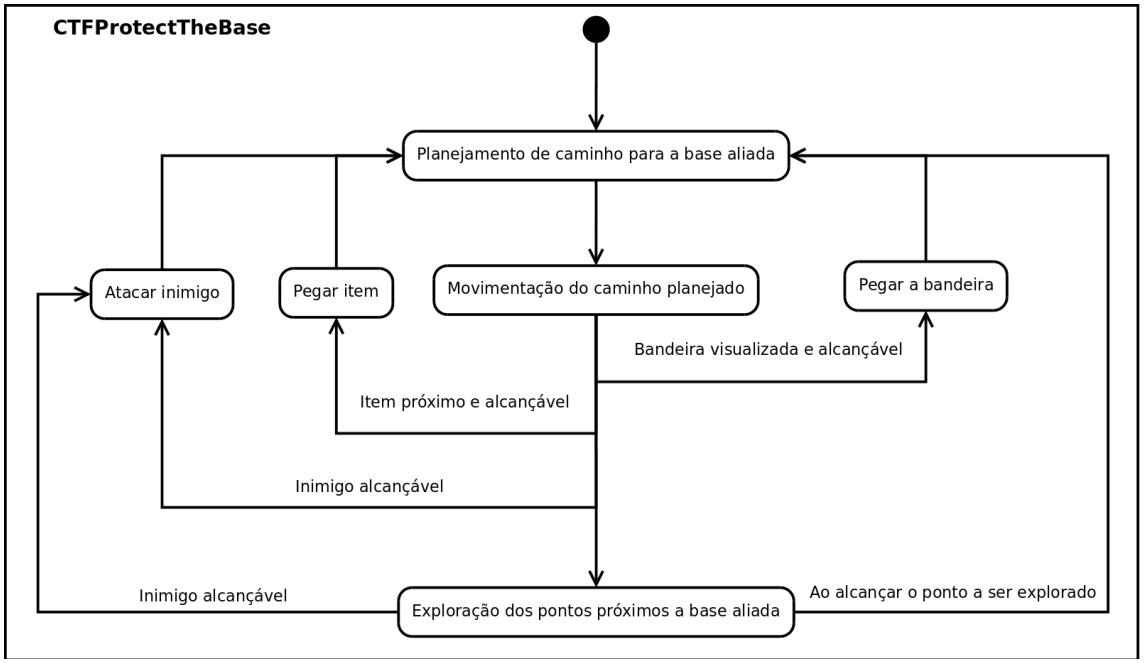

Figura 4.3: Papel Protect The Base.

Explore, puderam ser aproveitados no desenvolvimento dos *bots*, focando o desenvolvimento nas camadas executiva e deliberativa. Detalhes sobre o *framework* e a arquitetura estão presentes no Apêndice A.

A arquitetura de agente utilizada pelo *IAF* (MONTEIRO, 2005) fornece ao agente uma organização em três camadas: uma deliberativa, responsável por soluções globais para tarefas complexas, que no exemplo utilizado neste experimento faz a integração com o *proxy*; uma executiva, que seqüencia as tarefas passadas pela camada deliberativa, e um reativa, que é composta por comportamentos elementares que são selecionados pela camada executiva.

O mapeamento dos ambientes foi executado *offline*, gerando um arquivo de mapa a ser utilizado pelo agentes. A Listagem 4.1 é um exemplo de um arquivo de mapa de um cenário pequeno. Esses mapas eram compostos por coordenadas de itens no cenário, localização dos marcos (*waypoints*) utilizados pelos *bots* para a navegação, e a posição das bandeiras em cada base.

A maior parte do trabalho para o desenvolvimento dos *bots* ficou por conta da implementação de papéis a serem desempenhados pelos agentes. Tais papéis foram implementados como máquinas de estado na camada executiva do agente. Assim, a camada deliberativa recebia o papel a ser executado do *proxy* e repassava à camada executiva.

A combinação das máquinas de estado no nível executivo com as máquinas que implementam os comportamentos no nível reativo formavam uma máquina de estados hierárquica, uma vez que as ações do nível executivo representavam seleções de comportamentos reativos. Desta maneira, mesmo máquinas de estados simples conseguiam desempenhar um comportamento complexo, dada a possibilidade de representar as ações em níveis de abstrações diferentes.

Nas Figuras 4.3 e 4.4 estão representados os diagramas de dois dos cinco papéis imple-

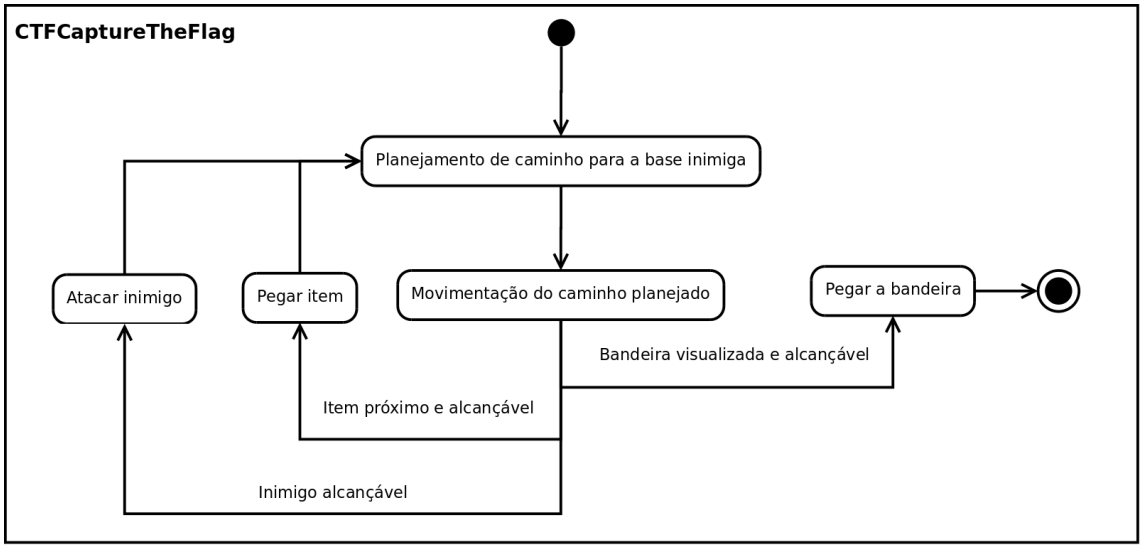

Figura 4.4: Papel Capture The Flag.

mentados, o papel de proteger a base (*CTFProtectTheBase*) e o de capturar a bandeira (*CTFCaptureTheFlag*). No papel de proteger a base, o agente inicialmente define o caminho que irá seguir para voltar até a base e então começa a movimentação sobre o caminho escolhido. Se no meio do caminho ele encontra um inimigo, percebe algum item[1] próximo ou visualiza alguma bandeira, então seu comportamento de movimentação muda para reagir à novidade. Caso ele visualize alguma bandeira, que pode ter sido perdida por algum jogador ou que pode estar sendo carregada por algum personagem, o agente segue em busca dessa bandeira, tentando pegá-la quando ela se encontra ao chão, resgatá-la quando se trata da bandeira aliada na mão de um inimigo, ou escoltá-la quando um aliado carrega a bandeira inimiga. Este comportamento persiste até que o agente consiga seu objetivo, até que seja morto pelo inimigo, ou ao perceber que a perseguição tornou-se irrelevante (ex. inimigo fez o ponto). No caso da visualização do inimigo e do item, o agente modifica sua movimentação apenas se uma dessas entidades estiver ao seu alcance. Quando o *bot* chega na base, ele inicia o processo de exploração, executando uma espécie de patrulha para defender a base aliada e preservar a bandeira. Após chegar ao ponto escolhido para exploração da base, o *bot* retorna à posição da bandeira para começar um novo ciclo de patrulha.

No papel de capturar a bandeira, mostrado na Figura 4.4, o agente escolhe um caminho para seguir até a base inimiga e então começa a sua busca pela bandeira. No caminho ele pode entrar em combate com um inimigo e pegar itens caso eles estejam ao seu alcance, priorizando sempre a perseguição de quem esteja com a bandeira. Sempre que ele termina uma perseguição, um combate ou um desvio para pegar um item, um novo caminho é escolhido até a base inimiga. Quando consegue chegar à base inimiga e capturar a bandeira, o objetivo foi concluído e o agente informa a situação ao seu *proxy* para que seja iniciado um novo processo de alocação de papéis.

As decisões de mais alto nível são as decisões tomadas pelos *proxies* que são repassadas para o agente através da camada deliberativa. Nesta camada deliberativa é implementada

[1]Os itens de cenários são compostos por munições, armas, kit médico, ou outro elemento presente no cenário que o *bot* consegue pegar.

a comunicação com o *proxy*, interpretando as ordens passadas e enviando atualizações de crenças. No exemplo utilizado no experimento, as crenças de domínio que são mantidas atualizadas pelo agente são o estado das duas bandeiras, a aliada e a inimiga. Sempre que o estado de uma bandeira se modifica, o agente que percebeu a modificação informa para o seu *proxy* sobre a alteração. Os agentes também respondem aos questionamentos do *proxy* sobre suas capacidades para executar determinados papéis. Eles calculam suas habilidades atuais para cada capacidade e as enviam para o *proxy*. Nos *bots* implementados para o experimento, a habilidade para executar um determinado papel é dada pela distância para as bandeiras, por exemplo, um personagem que está na base aliada está mais habilitado a proteger a base do que um personagem que se encontra na base inimiga.

Listagem 4.1: Exemplo de mapa fornecido ao agente.

```
NAV {Id CTF-1on1-Joust.InventorySpot7} {Location 1363.36,11252.86,-3201.00}
NAV {Id CTF-1on1-Joust.InventorySpot6} {Location 1368.08,10745.83,-3201.00}
NAV {Id CTF-1on1-Joust.InventorySpot5} {Location -1390.49,10750.54,-3201.00}
NAV {Id CTF-1on1-Joust.InventorySpot4} {Location -1390.55,11203.44,-3201.00}
NAV {Id CTF-1on1-Joust.PathNode3} {Location -972.16,10975.37,-3201.00}
NAV {Id CTF-1on1-Joust.PathNode2} {Location 3.84,10975.37,-3201.00}
NAV {Id CTF-1on1-Joust.PathNode1} {Location 547.84,10975.37,-3177.26}
NAV {Id CTF-1on1-Joust.PathNode0} {Location -404.16,11007.37,-3185.08}
NAV {Id CTF-1on1-Joust.PlayerStart8} {Location 1399.58,11944.15,-3201.00}
NAV {Id CTF-1on1-Joust.PlayerStart3} {Location -1408.21,9961.93,-3201.00}
NAV {Id CTF-1on1-Joust.PlayerStart0} {Location -1422.96,11909.79,-3201.00}
NAV {Id CTF-1on1-Joust.PlayerStart1} {Location 1405.89,9955.62,-3196.95}
NAV {Id CTF-1on1-Joust.xRedFlagBase0} {Location -1419.76,10976.04,-3148.00}
NAV {Id CTF-1on1-Joust.xBlueFlagBase0} {Location 1394.96,10975.91,-3149.00}
INV {Id CTF-1on1-Joust.RocketLauncherPickup} {Location -1390.55,11203.44,-3193.00}
INV {Id CTF-1on1-Joust.SniperRiflePickup} {Location -1390.49,10750.54,-3194.00}
INV {Id CTF-1on1-Joust.RocketLauncherPickup} {Location 1368.08,10745.83,-3194.00}
INV {Id CTF-1on1-Joust.SniperRiflePickup} {Location 1363.36,11252.86,-3194.00}
INV {Id CTF-1on1-Joust.AssaultRifle} {Location -1400.35,10827.25,-3198.15}
INV {Id CTF-1on1-Joust.ShieldGun} {Location -1408.21,9961.93,-3201.00}
INV {Id CTF-1on1-Joust.Translauncher} {Location -1408.21,9961.93,-3201.00}
INV {Id CTF-1on1-Joust.AssaultRifle} {Location -1391.68,11124.29,-3198.10}
INV {Id CTF-1on1-Joust.ShieldGun} {Location -1422.96,11909.79,-3201.00}
INV {Id CTF-1on1-Joust.Translauncher} {Location -1422.96,11909.79,-3201.00}
INV {Id CTF-1on1-Joust.AssaultRifle} {Location -1391.41,10709.97,-3198.15}
INV {Id CTF-1on1-Joust.ShieldGun} {Location -1408.21,9961.93,-3201.00}
INV {Id CTF-1on1-Joust.Translauncher} {Location -1408.21,9961.93,-3201.00}
INV {Id CTF-1on1-Joust.AssaultRifle} {Location -1392.03,11176.10,-3198.10}
INV {Id CTF-1on1-Joust.ShieldGun} {Location -1422.96,11909.79,-3201.00}
INV {Id CTF-1on1-Joust.Translauncher} {Location -1422.96,11909.79,-3201.00}
INV {Id CTF-1on1-Joust.AssaultRifle} {Location -1396.22,11326.82,-3198.10}
INV {Id CTF-1on1-Joust.ShieldGun} {Location -1422.96,11909.79,-3201.00}
INV {Id CTF-1on1-Joust.Translauncher} {Location -1422.96,11909.79,-3201.00}
INV {Id CTF-1on1-Joust.AssaultRifle} {Location -1396.61,10478.33,-3198.15}
INV {Id CTF-1on1-Joust.ShieldGun} {Location -1408.21,9961.93,-3201.00}
INV {Id CTF-1on1-Joust.Translauncher} {Location -1408.21,9961.93,-3201.00}
INV {Id CTF-1on1-Joust.RocketLauncher} {Location -1395.46,11006.51,-3182.10}
INV {Id CTF-1on1-Joust.SniperRifle} {Location -1415.24,10877.61,-3195.17}
INV {Id CTF-1on1-Joust.RocketLauncher} {Location -1390.55,11203.44,-3193.00}
INV {Id CTF-1on1-Joust.SniperRifle} {Location -1390.49,10750.54,-3194.00}
FLG {Id CTF-1on1-Joust.xBlueFlag} {Team CTF-1on1-Joust.TeamBlueConfigured}
    {State home} {Location 1394.96,10975.91,-3149.00}
FLG {Id CTF-1on1-Joust.xRedFlag} {Team CTF-1on1-Joust.TeamRedConfigured}
    {State home} {Location -1395.46,11006.51,-3182.10} {Holder CTF-1on1-Joust.xPawn}
```

O time que jogou *Capturar a Bandeira* usando o *TWProxy* possuía apenas quatro planos que eram ativados quando o estado da bandeira mudava e que podem ser descritos sucintamente como:

1. quando um time está com sua bandeira mas não está com a do inimigo, alguém precisa proteger a bandeira na base enquanto outros vão capturar a bandeira inimiga;

Tabela 4.2: Resultado das pontuações de time e individuais nas partidas de *Capturar a Bandeira* entre o *TWProxy-Team* e o *UTBots-Average*.

	TWProxy-Team		**UTBots-Average**	
Percentual de vitórias	$93,33\%$		0%	
Pontuação de Time	$\mu = 2,42$	$\sigma = 1,45$	$\mu = 0,03$	$\sigma = 0,18$
Soma das Pontuações Individuais	$\mu = 230,97$	$\sigma = 31,11$	$\mu = 100,77$	$\sigma = 20,27$

2. quando um time tem ambas as bandeiras, todos voltam para a base aliada;

3. quando um time não tem a bandeira aliada mas tem a bandeira inimiga, quem está com a bandeira inimiga volta para a base enquanto os outros tentam recuperar a bandeira aliada;

4. quando um time não possui nenhuma das bandeiras, todos seguem em busca de qualquer bandeira.

A fim de comparar o *TWProxy* com outra solução de *teamwork*, foi utilizado o *Machinetta* devido ao grande número de aplicações bem sucedidas em ambientes dinâmicos (SCERRI et al., 2003b; SCHURR et al., 2005b; SCERRI et al., 2003a,c; CHALUPSKY et al., 2001; SCERRI; PYNADATH; TAMBE, 2001, 2002). Para estes experimentos, todos os agentes do *TWProxy-Team* (time com agentes coordenados pelo *TWProxy*) e do *Machinetta-Team* (time com agentes coordenados pelo *Machinetta*) eram homogêneos, com mesma implementação e conseqüentemente mesmas capacidades. Assim, apenas o trabalho em equipe fazia a diferença no combate entre eles.

Para a avaliação da eficência de *teamwork* conseguida com o *TWProxy*, foram executados diversos experimentos de combates entre times distintos. Os times que se enfrentaram foram: *TWProxy-Team* vs *UTBots-Average* (time de bots do próprio jogo UT2004 no nível de habilidade *Average*), *TWProxy-Team* vs *UTBots-Experienced* (time de bots do próprio jogo UT2004 no nível de habilidade *Experienced*), *Machinetta-Team* vs *UTBots-Average*, *TWProxy-Team* vs *Machinetta-Team* e *TWProxy-Team* vs *Human-Team* (time de humanos voluntários jogadores de UT2004). Cada combate tinha a duração de quinze minutos e foram executados trinta combates para cada combinação de time que se enfrentava. Nesses combates, cada time possuía cinco jogadores, com excessão dos combates entre *TWProxy-Team* e *Human-Team* quando foram utilizados apenas quatro jogadores, o que facilitou a participação de uma quantidade maior de voluntários, com uma grade de horários mais flexível.

Nos experimentos realizados com o *Machinetta-Team*, o mesmo apresentou algumas limitações, como a impossibilidade de reusar planos terminados. Por conta disto, foi necessário criar muitos planos redundantes para manter os *bots* jogando. O número de planos pode ter afetado o seu desempenho, possivelmente sendo um dos culpados pelo alto uso de memória e processamento. Outro aspecto negativo apresentado pelo *Machinetta* foi a sua instabilidade. Além de variar bastante no tempo de resposta para uma alocação de papéis, muitas vezes o *Machinetta* simplesmente deixava de responder às atualizações de crenças.

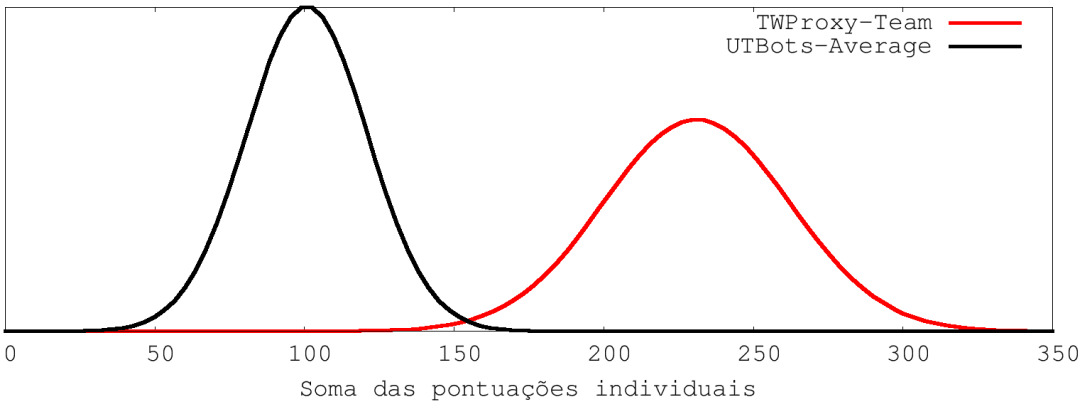

Figura 4.5: Guassianas da soma das pontuações individuais nos combates entre *TWProxy-Team* e *UTBots-Average*.

4.3.1 TWProxy-Team vs UTBots-Average

O experimento de combates entre os times *TWProxy-Team* e *UTBots-Average* foi composto por trinta batalhas com quinze minutos por combate. O *TWProxy-Team* era formado pelos agentes que utilizam o *TWProxy* enquanto o *UTBots-Average* era formado por *bots* do próprio jogo no nível de dificuldade *Average*, que é o nível de dificuldade padrão do jogo. Diferente do *TWProxy-Team*, o *UTBots-Average* possui um controle centralizado de seus jogadores e tem acesso a estados ocultos do jogo, como a posição dos inimigos e recursos disponíveis no ambiente. Ou seja, enquanto os agentes do *TWProxy-Team* estão limitados a uma observação parcial do ambiente, os *bots* do jogo têm acesso completo ao ambiente, fazendo maior uso ou menor uso dessas informações conforme se aumenta ou diminui a dificuldade do jogo.

O resultado das trinta partidas realizadas está sumarizado na Tabela 4.2, sob o formato de média (μ) e desvio padrão (σ) das pontuações de time e individuais. As médias e os desvios apresentados nessa tabela foram calculados com base nas pontuações obtidas nas partidas. A pontuação de time indica a quantidade de vezes que um time capturou a bandeira inimiga, enquanto que a soma da pontuação individual indica como se comportaram os membros da equipe. A pontuação individual dos jogadores é composta pela quantidade de inimigos abatidos mais bônus por bandeira capturada. A Figura 4.5 apresenta uma visualização qualitativa dos dados sobre as somas de pontuações individuais dos jogadores. Assim, é possível observar na Figura 4.5 que os jogadores do time *TWProxy-Team* tiveram melhor desempenho do que os jogadores do time *UTBots-Average*, e essa superioridade se confirmou também nos resultados finais dos combates, como pode ser visto na Tabela 4.2.

Apesar das vantagens dadas ao *UTBots-Average*, com o uso de controle centralizado e acesso a estados ocultos de ambiente, o *TWProxy-Team* ganhou $93,33\%$ das partidas e empatou apenas $6,67\%$. Considerando que o nível de dificuldade utilizado pelos *bots* do jogo (*Average*) é o padrão do jogo, é possível constatar com esse experimento que uma solução distribuída como o *TWProxy* tem condições de competir com uma abordagem totalmente centralizada em um domínio onde o compartilhamento de informações e a reação às atualizações é muito importante.

Tabela 4.3: Resultado das pontuações de time e individuais nas partidas de *Capturar a Bandeira* entre o *TWProxy-Team* e o *UTBots-Experienced*.

	TWProxy-Team		**UTBots-Experienced**	
Percentual de vitórias	$76,67\%$		0%	
Pontuação de Time	$\mu = 1,37$	$\sigma = 1,24$	$\mu = 0$	$\sigma = 0$
Soma das Pontuações Individuais	$\mu = 227,03$	$\sigma = 27,32$	$\mu = 117,87$	$\sigma = 27,72$

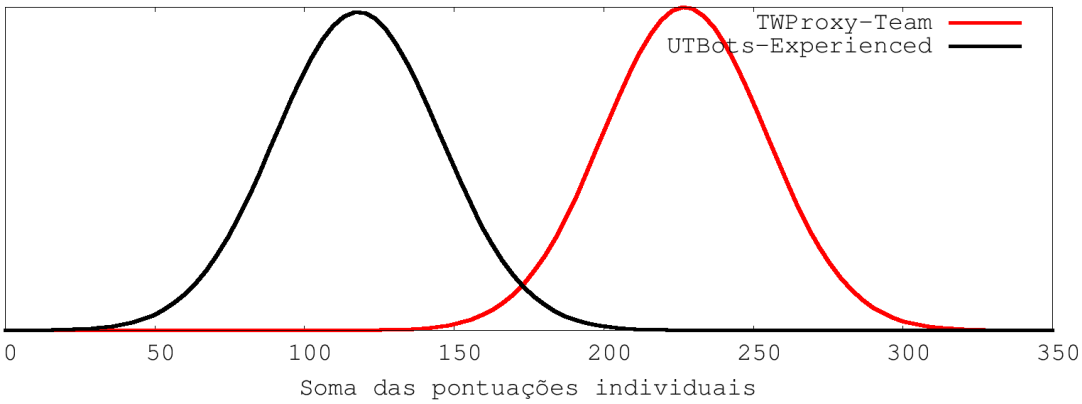

Figura 4.6: Guassianas da soma das pontuações individuais nos combates entre *TWProxy-Team* e *UTBots-Experienced*.

4.3.2 TWProxy-Team vs UTBots-Experienced

O experimento de combate entre os time *TWProxy-Team* e *UTBots-Experienced* é similar ao experimento anterior, onde o *TWProxy-Team* enfrenta *bots* do próprio jogo. Entretanto, neste experimento, o nível de dificuldade utilizado foi um nível acima do padrão (*Experienced*). Como o anterior, também foram realizadas trinta partidas com cinco jogadores em cada time.

Os resultados deste experimento são apresentados na Tabela 4.3, onde ainda é possível perceber uma grande diferença no desempenho individual dos jogadores, apesar das distâncias terem sido diminuídas com o aumento do nível dos *bots* do jogo, como pode ser percebido comparando a Figura 4.6 com a Figura 4.5. A diferença das somas individuais diminuíram $16,2\%$ com o aumento do nível dos *bots* em relação ao experimento anterior. Mesmo assim, nesse experimento o *UTBots-Experienced* não conseguiu capturar nenhuma bandeira do *TWProxy-Team*. No resultado geral deste experimento, o *TWProxy-Team* venceu $76,67\%$ e empatou $23,33\%$ das partidas.

4.3.3 Machinetta-Team vs UTBots-Average

Neste experimento, o *Machinetta-Team* enfrentou o *UTBots-Average* sobre as mesmas condições anteriores. O *Machinetta-Team* utilizou os mesmos agentes que atuaram para o time *TWProxy-Team* e a diferença entre eles ficou por conta apenas dos *proxies* de time. Esse experimento foi análogo ao experimento da subseção 4.3.1, mas os resultandos indicam existência de uma grande diferença entre o *Machinetta* e o *TWProxy* para este domínio.

Tabela 4.4: Resultado das pontuações de time e individuais nas partidas de *Capturar a Bandeira* entre o *Machinetta-Team* e o *UTBots-Average*.

	Machinetta-Team		**UTBots-Average**	
Percentual de vitórias	20%		0%	
Pontuação de Time	$\mu = 0,27$	$\sigma = 0,64$	$\mu = 0,03$	$\sigma = 0,18$
Soma das Pontuações Individuais	$\mu = 185,47$	$\sigma = 48,28$	$\mu = 73,87$	$\sigma = 31,84$

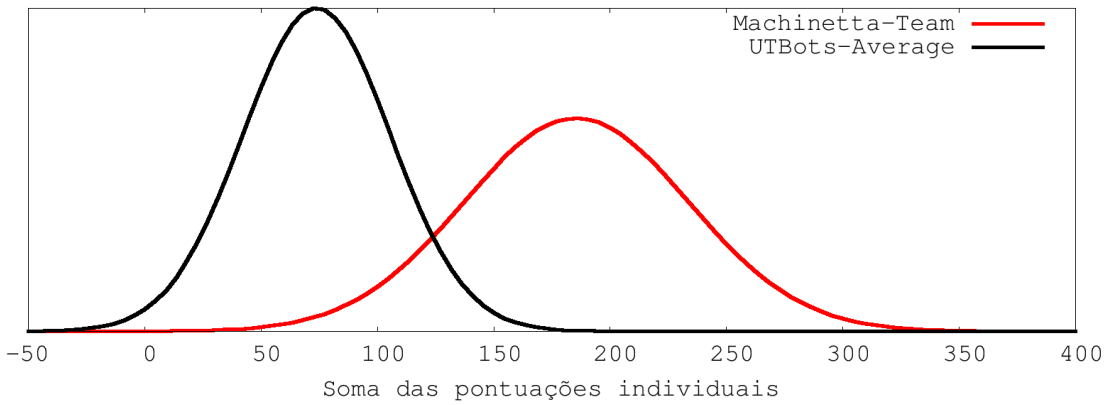

Figura 4.7: Guassianas da soma das pontuações individuais nos combates entre *Machinetta-Team* e *UTBots-Average*.

A Tabela 4.4 apresenta os resultados deste experimento, no qual os jogadores individuais do *Machinetta-Team* tiveram um desempenho melhor que os *bots* do próprio jogo, como pode ser visto também na Figura 4.7. Tal desempenho individual já era esperado, dado que os agentes são os mesmos utilizados pelo *TWProxy-Team*. Entretanto, o desempenho do time, apesar de ter sido um pouco melhor que o nível básico do jogo, foi muito menor que a eficiência conseguida pelo *TWProxy-Team* nas mesma condições, como pode ser visto na Tabela 4.2.

No resultado geral das partidas deste experimento, o *Machinetta-Team* venceu 20% e empatou 80% das partidas. Assim, é possível fazer um comparativo quantitativo entre o *TWProxy* e o *Machinetta* usando como base o experimento da subseção 4.3.1. Com uma queda significativa da eficiência do time de $89,84\%$ e uma queda na quantidade de vitória de $78,5\%$. É possível perceber o quanto uma ferramenta de coordenação pode influenciar no desempenho de um time. Esses resultados evidenciam que o *TWProxy* está mais preparado que o *Machinetta* para o domínio aqui apresentado.

4.3.4 TWProxy-Team vs Machinetta-Team

Após uma comparação indireta entre *TWProxy* e *Machinetta*, este experimento fornece os resultados necessários para uma comparação direta entre as duas ferramentas. Neste, foram executadas trinta batalhas entre o *TWProxy-Team* e o *Machinetta-Team*, onde os times usavam o mesmo conjunto de agentes e a diferença entre eles ficava por conta apenas do *proxy* de time utilizado.

O resultado das pontuações individuais foi razoavelmente equilibrado, como se pode ver na grande intersecção entre as gaussianas da Figura 4.8. Entretanto, os resultados indi-

Tabela 4.5: Resultado das pontuações de time e individuais nas partidas de *Capturar a Bandeira* entre o *TWProxy-Team* e o *Machinetta-Team*.

	TWProxy-Team		**Machinetta-Team**	
Percentual de vitórias	$43,33\%$		0%	
Pontuação de Time	$\mu = 0,6$	$\sigma = 0,81$	$\mu = 0$	$\sigma = 0$
Soma das Pontuações Individuais	$\mu = 217,27$	$\sigma = 57,15$	$\mu = 176,07$	$\sigma = 29,61$

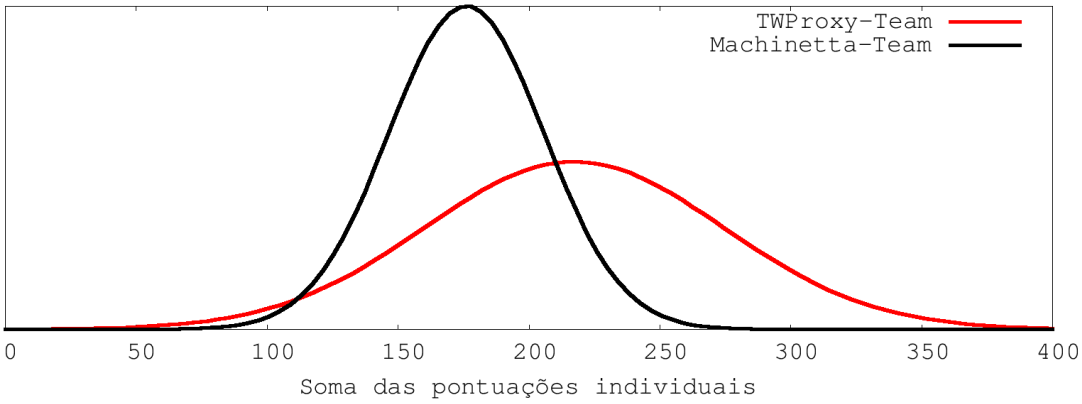

Figura 4.8: Guassianas da soma das pontuações individuais nos combates entre *TWProxy-Team* e *Machinetta-Team*.

viduais foram um pouco mais favoráveis ao *TWProxy-Team*, apesar do uso dos mesmos jogadores pelos dois times. A justificativa principal para essa pequena diferença são os bônus extras adquiridos com a captura da bandeira. Mesmo assim, o resultado ainda foi próximo.

O *Machinetta-Team* não conseguiu pontuar sobre o *TWProxy-Team*, enquanto o *TWProxy-Team* teve uma média de $0,6$ capturas sobre o *Machinetta-Team* (como pode ser visto na Tabela 4.5), Esta diferença de pontuação de time pode parecer pouco à primeira vista, entretanto, o resultado geral das partidas dá as reais dimensões desse valor. No resultado geral, $43,33\%$ das partidas resultaram em vitória do *TWProxy-Team* enquanto $56,67\%$ das partidas representaram empate entre as duas equipes. Com base na comparação direta desta subseção somada à comparação indireta nas subseções 4.3.1 e 4.3.3, é possível identificar que o *TWProxy* apresenta vantagens claras sobre o *Machinetta* para o domínio utilizado, que representa um ambiente dinâmico com requisitos de tempo-real.

Tabela 4.6: Resultado das pontuações de time e individuais nas partidas de *Capturar a Bandeira* entre o *TWProxy-Team* e o *Human-Team*.

	TWProxy-Team		**Human-Team**	
Percentual de vitórias	20%		$26,67\%$	
Pontuação de Time	$\mu = 0,23$	$\sigma = 0,43$	$\mu = 0,5$	$\sigma = 0,97$
Soma das Pontuações Individuais	$\mu = 108,73$	$\sigma = 25,02$	$\mu = 154,2$	$\sigma = 39,23$

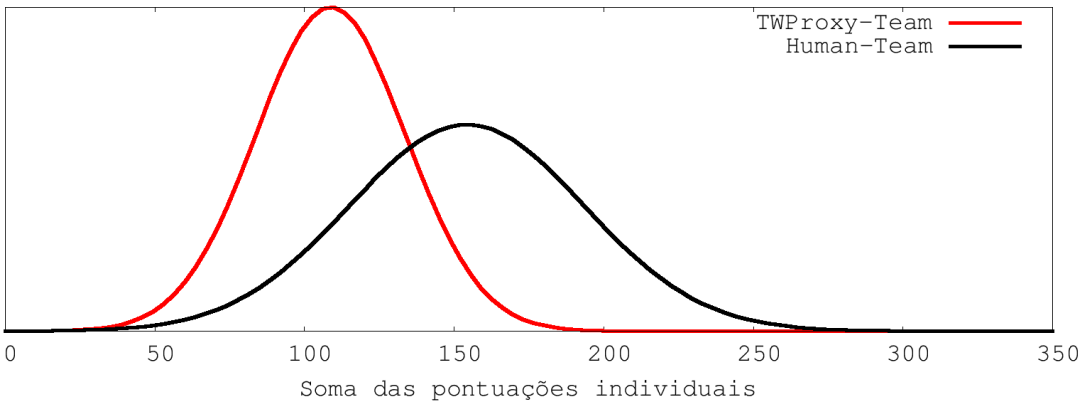

Figura 4.9: Guassianas da soma das pontuações individuais nos combates entre *TWProxy-Team* e *Human-Team*.

4.3.5 TWProxy-Team vs Human-Team

Neste último experimento de combate buscou-se avaliar a coerência de *teamwork* oferecida pelo *TWProxy* colocando o *TWProxy-Team* para jogar contra times de jogadores humanos (*Human-Team*). Cada time possuía quatro jogadores e as partidas duravam quinze minutos. Vinte e cinco voluntários, jogadores de UT2004, se revezaram para formar os times de humanos, e entre esses voluntários era muito diversificado o nível de habilidade para jogar UT2004.

A principal motivação para realizar os experimentos com humanos era identificar as possíveis incoerências a que o time poderia estar se submetendo e que jogadores humanos experientes poderiam explorar com maior facilidade. Tais falhas no time de agentes pode não ser identificada em um resultado final com pontuações de jogos, por isso, os comentários dos voluntários também foram levados em consideração para avaliar se os problemas identificados eram da coordenação do time ou da implementação do agente.

Uma observação inicial feita pelos voluntários foi que os *bots* utilizam muito bem as armas de projéteis instantâneos, como rifles e pistola, enquanto usam muito mal armas como lançadores de granada. Este fato se deve a como foi implementada a mira dos *bots*, que não fornece uma predição sobre um alvo em movimento, mas sim um posicionamento direto para a coordenada especificada.

Uma característica interessante dos times humanos é que eles evoluíam com o tempo, identificando as estratégias dos *bots* e adaptando as suas próprias estratégias. Alguns times de humanos perceberam que o agente não explorava determinados pontos do mapa e passavam a usar tais pontos como posições estratégicas. Outros times sabiam aproximadamente onde deveriam estar os *bots* que defendiam a base e já entravam atirando no local previsto.

Mesmo assim, as partidas foram bastante disputadas, como pode ser visto na Tabela 4.6. Dos trinta jogos disputados, o *TWProxy-Team* ganhou 20%, perdeu $26,67\%$ e empatou $53,33\%$. A Figura 4.9 mostra de forma qualitativa o quão disputadas foram as partidas, com jogadores humanos atuando em média melhor que os *bots* do *TWProxy-Team*, mas ainda assim travando combates equilibrados, como pode ser visto pela sobreposição das gaussianas que representam a soma das pontuações individuais.

A maioria dos times de humanos que venceram as partidas convergiram para uma mesma estratégia. Um jogador assumia o papel de atirador de elite e ficava em cima de uma torre, onde os *bots* não conseguiam vê-lo. De lá, ele defendia toda a equipe atirando nos pontos onde surgiam os *bots*, evitando que os inimigos entrassem na própria base para defender a bandeira. Enquanto isso, outros três jogadores humanos seguiam para capturar a bandeira. Nessa estratégia, normalmente apenas um jogador conseguia chegar à base inimiga e pegar a bandeira, uma vez que a defesa dos bots era muito boa. Assim que um jogador humano pegava a bandeira inimiga, escondia-se em pontos do cenário que eles sabiam que os *bots* dificilmente vasculhariam. Enquanto ele se preparava para voltar para a base aliada, o atirador de elite dava cobertura em campo aberto e os que haviam morrido, avançavam para dar cobertura de perto após ressurgir no cenário.

As explorações dos comportamentos do *TWProxy-Team* demonstraram que os principais problemas para o time de *bots* foram: o fato dos agentes não serem adaptativos, não armazenar nem fazer uso de informações sobre pontos estratégicos do cenário e de não ter sido implementado um papel de atirador de elite. Adicionalmente, nenhuma das derrotas do *TWProxy-Team* remeteram a uma falha direta na coordenação dos agentes, na realidade, os resultados obtidos mostram que este modelo de coordenação é bastante flexível e robusto, permitindo que agentes não-adaptativos atuem de forma bastante competitiva contra jogadores humanos experientes.

5 CONSIDERAÇÕES FINAIS

Teamwork tem se tornado um paradigma padrão para a coordenação de agentes de forma cooperativa, mostrando-se robusto e flexível para domínios complexos como, robótica móvel, automação industrial, controle aéreo, simulação de resgates, entre outros. Com o desenvolvimento de ferramentas de suporte a *teamwork*, como o *STEAM* e o *Machinetta*, tornou-se possível a reutilização do modelo de *teamwork* através de diferentes domínios. Entretanto, as ferramentas pré-existentes não fornecem uma solução geral para todos os domínios, principalmente em se tratando de domínios que possuem ambientes altamente dinâmicos com requisitos de tempo real.

Com a intenção de viabilizar o uso do paradigma de *teamwork* em ambientes altamente dinâmicos, este trabalho introduz o *TWProxy*, uma nova ferramenta que utiliza abordagem de *proxy* de time para a coordenação das equipes. Com esta ferramenta é demonstrada a viabilidade do uso de *teamwork* para coordenação de agentes situados em ambientes altamente dinâmicos com requisitos de tempo-real. Com ela também se confirma a flexibilidade e a robustez do planejamento em alto-nível, juntamente com as facilidades em reuso do modelo de *teamwork* através da abordagem de *proxy*.

Através do *TWProxy* são introduzidas novas características importantes para o desenvolvimento de time em ambientes complexos. Nele são fornecidos dois algoritmos parcialmente distribuídos para alocação de tarefas, incluíndo um novo algoritmo heurístico com eficiência muito próxima da ótima. Também é disponibilizada uma linguagem simples e extensível para a descrição de crenças e planos, que fornece maior flexibilidade e autonomia ao time para executar suas tarefas. Além disso, permite a reutilização de planos e flexibilidade dos mesmos poderem ser executados por uma quantidade variável de agentes. Os agentes também possuem acesso a um quadro-negro distribuído que o *TWProxy* oferece devido à manutenção das crenças entre os *proxies*. Vale ressaltar que a utilização do *TWProxy* em novos domínios é bastante simplificada, com a necessidade de se escrever apenas o programa de time e a interface com o agente.

Os experimentos mostram ser viável o uso de *teamwork* em ambientes altamente dinâmicos, além de comprovar as vantagens do *TWProxy* frente ao *Machinetta*. Nesses experimentos, o *TWProxy* foi avaliado principalmente sobre os seguintes aspectos: a qualidade da alocação de papéis, o período em que o time se mantém em estado inconsistente e a eficiência que esses dois aspectos anteriores podem fornecer ao time.

O *TWProxy* teve bons resultados, sendo caracterizado de maneira geral como estável e eficiente. Na avaliação da qualidade da alocação de papéis, um dos algoritmos é ótimo em tempo polinomial, enquanto o outro é heurístico com resultados muito próximos do ótimo

e com menor complexidade que o primeiro. Na avaliação do período de inconsistência de time, o *TWProxy* é comparado com o *Machinetta* em um ambiente simulado, onde o *TWProxy* obtém períodos de inconsistência muito menores que o *Machinetta*. Quanto a eficiência, avaliada sobre uma aplicação real, foi possível constatar que o *TWProxy* consegue manter a coerência do time mesmo em um ambiente altamente dinâmico, competindo com abordagens centralizadas e contra times de humanos. As comparações com o *Machinetta* também foram importantes para demonstrar a variação de eficiência que um time pode sofrer devido à mudança da sua ferramenta de coordenação.

Apesar dos avanços obtidos com o *TWProxy*, a ferramenta aqui apresentada necessita de algumas funcionalidades e características importantes para tornar-se mais geral no suporte a *teamwork* em ambientes altamente dinâmicos. Por isso, dentre os próximos passos estão: a implementação do suporte a novas organizações de agentes; o desenvolvimento de uma camada de comunicação tolerante a falhas, tentando minimizar o impacto do atraso inserido por mensagens perdidas; e a investigação sobre a autonomia ajustável em ambientes altamente dinâmicos, com agentes semi-autônomos que permitam a participação de humanos em times híbridos.

REFERÊNCIAS

BYL, P. B. de. **Programming Believable Characters for Computer Games**. Hingmam, USA: Charles River, 2004. 465p.

CHALUPSKY, H. et al. Electric elves: applying agent technology to support human organizations. In: IAAI, 2001, Menlo Park, USA. **Proceedings...** [S.l.: s.n.], 2001. p.51–58.

DEFAGO, X.; SCHIPER, A.; URBAN, P. Total order broadcast and multicast algorithms: taxonomy and survey. **ACM Comput. Surv.**, New York, NY, USA, v.36, n.4, p.372–421, 2004.

EBERLY, D. H. **3D Game Engine Design**. San Francisco, USA: Morgan Kaufmann, 2000. 561p.

FININ, T. et al. KQML as an Agent Communication Language. In: INTERNATIONAL CONFERENCE ON INFORMATION AND KNOWLEDGE MANAGEMENT, 3., 1994, Gaithersburg, MD, USA. **Proceedings...** New York: ACM Press, 1994. p.456–463.

GROSZ, B. J.; KRAUS, S. Collaborative Plans for Complex Group Action. **Artificial Intelligence**, [S.l.], v.86, n.2, p.269–357, 1996.

GROSZ, B. J.; KRAUS, S. The Evolution of SharedPlans. In: WOOLDRIDGE, M.; RAO, A. (Ed.). **Foundations and Theories of Rational Agency**. Boston, USA: Kluwer Academic Publishers, 1999. p.227–262.

GRüNVOGEL, S. M. Dynamic Character Animation. **International Journal of Intelligent Games and Simulation.**, [S.l.], v.2, n.1, p.11–19, 2003.

HILL, R. et al. Intelligent agents for the synthetic battlefield: A company of rotary wing aircraft. In: INNOVATIVE APPLICATIONS OF ARTIFICIAL INTELLIGENCE, 1997. **Proceedings...** [S.l.: s.n.], 1997.

JENNINGS, N. R. Controlling Cooperative Problem Solving in Industrial Multi-Agent Systems Using Joint Intentions. **Artificial Intelligence**, Essex, UK, v.75, n.2, p.195–240, 1995.

JONES, R. M. et al. Automated Intelligent Pilots for Combat Flight Simulation. **AI Magazine**, [S.l.], v.20, n.1, p.27–41, 1999.

KAMINKA, G. A. The RoboCup-98 Teamwork Evaluation Session: a preliminary report. In: ROBOCUP, 1999. **Proceedings...** [S.l.: s.n.], 1999. p.345–356.

KAMINKA, G. A. et al. GameBots: a flexible test bed for multiagent team research. **Commun. ACM**, New York, NY, USA, v.45, n.1, p.43–45, 2002.

KUHN, H. W. The Hungarian method for the assignment problem. **Naval Research Logistic Quarterly**, [S.l.], v.2, p.83–97, 1955.

LAIRD, J. E.; NEWELL, A.; ROSENBLOOM, P. S. SOAR: an architecture for general intelligence. **Artificial Intelligence**, Essex, UK, v.33, n.1, p.1–64, 1987.

LEVESQUE, H. J.; COHEN, P. R.; NUNES, J. H. T. On Acting Together. In: AAAI, 1990, Boston, USA. **Proceedings...** [S.l.: s.n.], 1990. p.94–99.

LEWIS, R. L. **Cognitive Theory, SOAR**. Disponível em: <http://www-personal.umich.edu/%7erickl/Documents/cognitive-theory-soar.pdf>. Acesso em: fev. de 2009.

MARSELLA, S. et al. On being a teammate: experiences acquired in the design of RoboCop teams. In: INTERNATIONAL CONFERENCE ON AUTONOMOUS AGENTS, 1999, Seattle, USA. **Proceedings...** New York: ACM Press, 1999. p.221–227.

MARSELLA, S. et al. Experiences Acquired in the Design of RoboCup Teams: a comparison of two fielded teams. **Autonomous Agents and Multi-Agent Systems**, [S.l.], v.4, n.1-2, p.115–129, 2001.

MONTEIRO, I. M. Uma Arquitetura Modular para o Desenvolvimento de Agentes Cognitivos em Jogos de Primeira e Terceira Pessoa. In: WORKSHOP BRASILEIRO DE JOGOS E ENTRETENIMENTO DIGITAL, 2005, São Paulo, Brasil. **Proceedings...** [S.l.: s.n.], 2005. p.219–229.

MONTEIRO, I. M.; ALVARES, L. O. A Real-Time Proxy for Flexible Teamwork in Dynamic Environments. In: BRAZILIAN SYMPOSIUM ON COMPUTER GAMES AND DIGITAL ENTERTAINMENT - COMPUTING TRACK, 2008, Belo Horizonte, Brasil. **Proceedings...** [S.l.: s.n.], 2008. p.83–90.

MONTEIRO, I. M.; SANTOS, D. A. dos. Um Framework para o Desenvolvimento de Agentes Cognitivos em Jogos de Primeira Pessoa. In: BRAZILIAN SYMPOSIUM ON COMPUTER GAMES AND DIGITAL ENTERTAINMENT - COMPUTING TRACK, 2007, São Leopoldo, RS, Brasil. **Proceedings...** [S.l.: s.n.], 2007. p.107–115.

MUNKRES, J. Algorithms for the Assignment and Transportation Problems. **Journal of the Society for Industrial and Applied Mathematics**, [S.l.], v.5, n.1, p.32–38, 1957.

NAIR, R. et al. RoboCup-Rescue: a proposal and preliminary experiences. In: ROBOCUP-RESCUE WORKSHOP; INTERNATIONAL CONFERENCE ON MULTI-AGENT SYSTEMS, 2000. **Proceedings...** [S.l.: s.n.], 2000.

PYNADATH, D. V.; TAMBE, M. An Automated Teamwork Infrastructure for Heterogeneous Software Agents and Humans. **Autonomous Agents and Multi-Agent Systems**, Hingham, MA, USA, v.7, n.1-2, p.71–100, 2003.

RICH, C.; SIDNER, C. L. COLLAGEN: when agents collaborate with people. In: INTERNATIONAL CONFERENCE ON AUTONOMOUS AGENTS, 1997, Manna Del Rey, CA. **Proceedings...** New York: ACM Press, 1997. p.284–291.

SCERRI, P. et al. **Getting Robots, Agents and People to Cooperate**: an initial report. Disponível em: <htpp://citeseer.ist.psu.edu/564721.html>. Acesso em: fev. de 2009.

SCERRI, P. et al. A prototype infrastructure for distributed robot-agent-person teams. In: AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS, 2003, Melbourne, Australia. **Proceedings...** New York: ACM Press, 2003. p.433–440.

SCERRI, P. et al. Team oriented programming and proxy agents: the next generation. In: INTERNATIONAL WORKSHOP ON PROGRAMMING MULTI-AGENT SYSTEM, 2003. **Proceedings...** Berlin: Springer, 2003. p.131–148.

SCERRI, P.; PYNADATH, D.; TAMBE, M. Adjustable autonomy in real-world multi-agent environments. In: INTERNATIONAL CONFERENCE ON AUTONOMOUS AGENTS, 2001, Montreal, Canada. **Proceedings...** New York: ACM Press, 2001. p.300–307.

SCERRI, P.; PYNADATH, D. V.; TAMBE, M. Towards adjustable autonomy for the real world. **Journal of Artificial Intelligence Research**, [S.l.], v.17, p.171–228, 2002.

SCHURR, N. et al. Evolution of a Teamwork Model. In: SUN, R.; SUN, R. (Ed.). **Cognition and Multi-Agent Interaction**: from cognitive modeling to social simulation. New York: Cambridge University Press, 2005. p.307–327.

SCHURR, N. et al. Towards flexible coordination of human-agent teams. **Multiagent Grid Syst.**, Amsterdam, NLD, v.1, n.1, p.3–16, 2005.

SCHWAB, B. **AI Game Engine Programming**. Hingman, USA: Charles River Media, 2004. 624p.

SMITH, R. G. The Contract Net Protocol: high-level communication and control in a distributed problem solver. **IEEE Transactions on Computers**, [S.l.], v.C-29, n.12, p.1104–1113, 1981.

STONE, P.; VELOSO, M. M. Task Decomposition and Dynamic Role Assignment for Real-Time Strategic Teamwork. In: INTERNATIONAL WORKSHOP ON AGENT THEORIES, ARCHITECTURES, AND LANGUAGES, 1998, Paris, France. **Proceedings...** Berlin: Springer-Verlog, 1998. p.293–308.

SYCARA, K. et al. **The RETSINA MAS Infrastructure**. Pittsburgh, PA: Robotics Institute, Carnegie Mellon University, 2003. (CMU-RI-TR-01-05). Disponível em: <http://citeseer.ist.psu.edu/article/sycara01retsina.html>. Acesso em: fev. de 2009.

SYCARA, K.; SUKTHANKAR, G. **Literature Review of Teamwork Models**. Pittsburgh, PA: Robotics Institute, Carnegie Mellon University, 2006. (CMU-RI-TR-06-50). Disponível em: <http://www.ri.cmu.edu/pub_files/pub4/sycara_katia_2006_1/sycara_katia_2006_1.pdf>. Acesso em: fev. de 2009.

TAMBE, M. Towards Flexible Teamwork. **Journal of Artificial Intelligence Research**, [S.l.], v.7, p.83–124, 1997.

TAMBE, M. Agent architectures for flexible, practical teamwork. In: NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE, 1997. **Proceedings...** [S.l.: s.n.], 1997. p.22–28.

TAMBE, M. et al. Teamwork in cyberspace: using teamcore to make agents team-ready. In: AAAI SPRING SYMPOSIUM ON AGENTS IN CYBERSPACE, 1999. **Proceedings...** Menlo Park: AAAI Press, 1999.

TAMBE, M.; ZHANG, W. Towards flexible teamwork in persistent teams: extended report. **Journal of Autonomous Agents and Multi-Agent Systems**, [S.l.], v.3, p.159–183, 1998.

TAMBE, M.; ZHANG, W. Towards Flexible Teamwork in Persistent Teams: extended report. **Autonomous Agents and Multi-Agent Systems**, [S.l.], v.3, n.2, p.159–183, 2000.

TOZOUR, P. Search Space Representations. In: AI GAME PROGRAMMING WISDOM 2, 2004. **Proceedings...** [S.l.]: Charles River Media, 2004. p.85–102.

VU, T. et al. MONAD: a flexible architecture for multi-agent control. In: AUTONOMOUS AGENTS AND MULTI-AGENT SYSTEMS, 2003. **Proceedings...** [S.l.: s.n.], 2003. p.449–456.

YEN, J. et al. CAST: collaborative agents for simulating teamwork. In: INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE, 2001. **Proceedings...** [S.l.: s.n.], 2001. p.1135–1144.

APÊNDICE A ARQUITETURA DOS AGENTES PARA UT2004

Este apêndice descreve a arquitetura e *framework* utilizados para o desenvolvimento dos bots para experimento com o jogo Unreal Tournament 2004. Inicialmente é apresentada a arquitetura de agente utilizada (MONTEIRO, 2005), detalhando seus módulos e fluxos de informações, e em seguida é feita uma breve descrição do *framework* utilizado (MONTEIRO; SANTOS, 2007), apresentando alguns dos seus principais recursos.

A.1 Arquitetura

Os agentes desenvolvidos para o jogo UT2004 utilizam uma arquitetura modular para agentes cognitivos especializada para jogos de primeira pessoa (MONTEIRO, 2005). Tal arquitetura é composta por sete módulos, como pode ser visto na Figura A.1 e cada módulo desempenha um papel específico como é descrito a seguir.

- **Perception -** Módulo de eventos responsável pela filtragem das informações sensoriais do agente.
- **Location -** Centraliza informações sobre o modelo do ambiente para o agente.
- **Long-Term Decision -** Responsável por ações de alto-nível, incluíndo planejamento.
- **Short-Term Decision -** Responsável pela execução de um determinado papel.
- **Motion -** Manipula aspectos de roteamento, colisão e desvios de obstáculos.
- **Animation -** Módulo responsável por determinar a animação do personagem a ser exibida.
- **Behavior -** Módulo responsável pela atuação sobre o ambiente de forma reativa.

Dos sete módulos apresentados, cinco são responsáveis pelo processo de decisão do agente, são eles: *Long-Term Decision*, *Short-Term Decision*, *Motion*, *Animation*, *Behavior*. Os módulos de decisão estão agrupados em 3 camadas:

- **Camada Deliberativa -** Responsável por gerar soluções globais para tarefas complexas, composta pelo módulo *Long-Term Decision*.
- **Camada Executiva -** Responsável pelo seqüenciamento do plano passado pela camada deliberativa, composta pelos módulos *Short-Term Decision* e *Motion*.

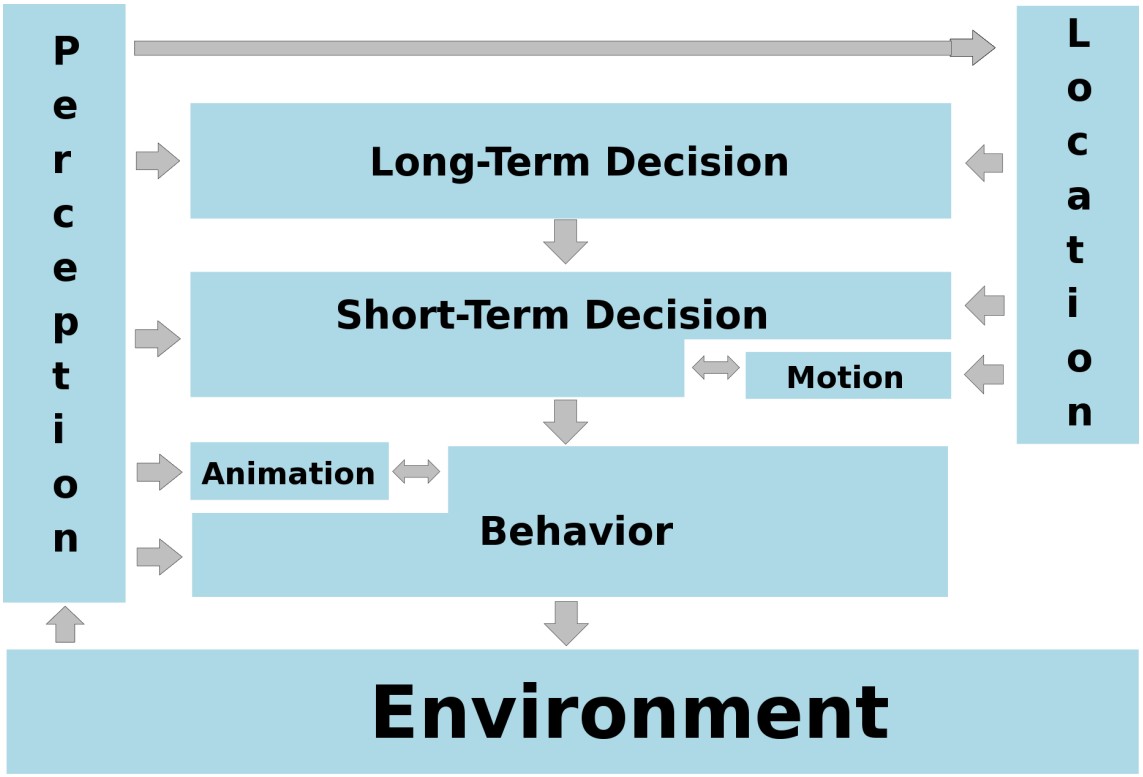

Figura A.1: A arquitetura.

- **Camada Reativa -** Responsável por responder às percepções do ambiente, composta pelos módulos *Animation* e *Behavior*.

O fluxo de informações através dos módulos é indicado na Figura A.1. Os dados captados do ambiente seguem para o módulo *Perception* que os filtra e os envia para os módulos responsáveis. *Perception*, após a filtragem, envia informações para *Location*, *Behavior*, *Animation*, *Short-Term Decision* e *Long-Term Decision*. Com a informação recebida de *Perception*, *Location* atualiza o modelo de ambiente do agente e disponibiliza, de forma síncrona com *Perception*, consultas a este modelo de ambiente para os módulos: *Long-Term Decision*, *Short-Term Decision* e *Motion*. Baseado no modelo atualizado do ambiente e nas percepções, *Long-Term Decision* utiliza sua representação interna de conhecimento para gerar planos que atendam aos objetivos do agente. Estes planos são passados para o *Short-Term Decision* que fica responsável pela execução dos mesmos. Para isto, *Short-Term Decision* utiliza as percepções e o modelo atualizado do ambiente juntamente com o módulo *Motion*, para decidir como executar cada tarefa de um dado plano. A execução das tarefas é feita trocando o comportamento reativo atual do módulo *Behavior*. Assim, *Behavior* atua sobre o ambiente baseado em suas percepções e o comportamento ativo atual. *Animation* e *Motion* são sub-módulos especializados de *Behavior* e *Short-Term Decision*, respectivamente. *Animation* comunica-se com o *Behavior* para decidir qual animação exibir e pode também receber percepções para gerar novas animações. Já o *Motion* é usado para decidir sobre a navegação do agente no ambiente.

Todos os dados captados por sensores do agente são tratados como percepções. Estes sensores podem ser dos mais variados tipos, imitando alguns dos sentidos humanos, sensores de visão, audição, tato e olfato. Cada percepção recebida deve ser repassada para

os módulos apropriados. Assim, *Perception* centraliza o recebimento de percepções, cria um modelo de representação interna dessas percepções e filtra o envio dessas para cada módulo. Essa filtragem é importante para fazer com que cheguem nos módulos apenas informações relevantes. Por exemplo, o som ambiente de uma forte ventania gera uma percepção auditiva que apesar de não ser muito útil para *Behavior*, pode ser interessante para *Long-Term Decision* concluir que vem chegando uma tempestade.

O módulo *Location* possui uma representação do ambiente de modo a facilitar questões como determinação de rotas, navegação, desvio de obstáculos e localização de objetos no ambiente. Em jogos, é comum gerar essa representação de maneira pré-processada, diminuindo assim, o processamento em tempo de execução. Com isto, para o agente, *Location* atua como um grande oráculo sobre o ambiente, sendo atualizado diretamente por conjuntos de percepções.

Long-Term Decision é o módulo responsável pelas decisões de mais alto nível para o agente. Este módulo prevê a utilização de modelos de tomada de decisão para gerar soluções globais para tarefas complexas. A principal atribuição para este nível é o planejamento. Assim, *Long-Term Decision* fica responsável por decidir qual a meta atual e o melhor plano para atingir esta meta, enviando a atividade que deve ser executada para *Short-Term Decision*, que fica encarregado de executá-la. Com isto, *Long-Term Decision* concentra-se em tipos de soluções estratégicas, delegando a execução dos planos para a camada inferior.

Short-Term Decision atua como integrador das decisões tomadas pelo *Long-Term Decision* e as ações executadas pelo *Behavior*. Dado um papel passado para *Short-Term Decision*, este módulo fica responsável por decidir como executar o papel recebido, e também possui a liberdade de acrescentar à atividade tarefas que otimizem o estado do agente e não desviem do objetivo do papel. A execução de tarefas é feita a partir da seleção de comportamentos no módulo *Behavior*. Dada uma seqüência de tarefas a ser executada, *Short-Term Decision* decide qual o comportamento ativar para a execução de cada uma delas. Uma tarefa pode conter um ou mais comportamentos seqüenciados. Dessa forma, *Short-Term Decision* atua sobre *Behavior* como um seletor do comportamento atual.

Motion é o responsável por entender como o agente deve se movimentar no mundo. Ele recebe informação de *Short-Term Decision* indicando para onde deve se movimentar e então decide a forma apropriada de se mover para o destino. Este módulo não executa a movimentação, apenas determina como executar. Toda decisão tomada por este módulo é computada com base nas estruturas de *Location*.

Animation é responsável por controlar o corpo do *bot* decidindo qual animação será apresentada para demonstrar o estado atual de ação deste persoagem. Estas animações são muitas vezes pré-geradas ou por animadores profissionais ou por captura de movimento, entretanto é possível gerar animação em tempo de execução(GRüNVOGEL, 2003). Um exemplo de geração de animação em tempo de execução é a utilização de cinemática inversa(EBERLY, 2000). O módulo de animação não é utilizado para o desenvolvimento de agentes para jogo UT2004 porque o jogo não fornece controle da animação do personagem, cuidando disso por conta própria.

Behavior é o módulo responsável por executar a ação sobre o ambiente. Dado que *Short-Term Decision* selecionou um comportamento, este passa a ser o comportamento reativo atual. O comportamento reativo selecionado capta as percepções do ambiente e executa

ações sobre o mesmo. O conjunto de ações do agente é similar às ações executadas pelo jogador humano através dos dispositivos de entrada.

A.2 Framework

O *framework* utilizado para o desenvolvimento dos agentes é o IAF (MONTEIRO; SANTOS, 2007). Ele é organizado em pacotes onde são agrupadas as diversas funcionalidades para dar suporte à criação de um agente para o jogo UT2004. O *framework* fornece o esqueleto para o desenvolvimento de *bots* para UT2004, incluíndo funcionalidades relacionadas à comunicação com *GameBots*, implementação da arquitetura anteriormente descrita e alguns comportamentos básicos implementados.

A.2.1 Comunicação com jogo

O *IAF* fornece todo o suporte à comunicação entre agente e jogo, que ocorre através da troca de mensagens via socket TCP. Neste suporte está incluída desde a conexão através de socket, até a transformação da mensagem em um tipo abstrato de dado[1]. Desta forma, as mensagens passam a ser utilizadas pelo *framework* como informação e não mais como dados.

A fábrica de mensagens sensoriais, o SensoryMessageFactory, é responsável por, dado uma mensagem em texto plano, no formato TYPE Arg1 Values1 ... ArgN ValuesN, gerar informação útil para o *framework*, ou seja, criar uma instância referente ao tipo de mensagem, com as operações necessárias para o acesso da mesma. Para isso, a fábrica realiza o *parser* das mensagens e decide qual instância de mensagen sensorial deve criar. As mensagens criadas são todas filhas de SensoryMessage, e elas se dividem em dois tipos básicos que são as mensagens síncronas e assíncronas.

As mensagens síncronas são recebidas pelo agente em um intervalo configurável e informam essencialmente o que o *bot* vê, o estado do mesmo e a pontuação no jogo. Já as mensagens assíncronas informam eventos que podem ocorrer a qualquer instante no jogo, como: ser atingido, comunicação textual, mudança de ambiente, colisão com alguma parede ou jogador, dano tomado, entre outros.

A.2.2 Implementação da Arquitetura

O pacote *architecture* do *IAF* fornece a implementação da estrutura proposta pela arquitetura definida em (MONTEIRO, 2005). Neste pacote, são organizados cada um dos módulos da arquitetura juntamente com seus relacionamentos. Uma importante classe deste pacote é a classe *Agent* que contém a estrutura de um agente cognitivo que implementa a arquitetura proposta. Esta classe *Agent* é de fácil extensão o que permite sua reutilização também para evoluções que venham ocorrer na arquitetura. Ela implementa todas as funcionalidades necessárias para se criar um *bot*, o que inclui toda a parte de comunicação com o jogo. A seguir são descritos os módulos implementados.

Utilização da Arquitetura: A classe *Agent* é a classe-base para o desenvolvimento de um agente no IAF. Ela define todos módulos da arquitetura e sua adaptação é feita através

[1]Tipo usado para encapsular outros tipos de dados e que possuem operações associadas

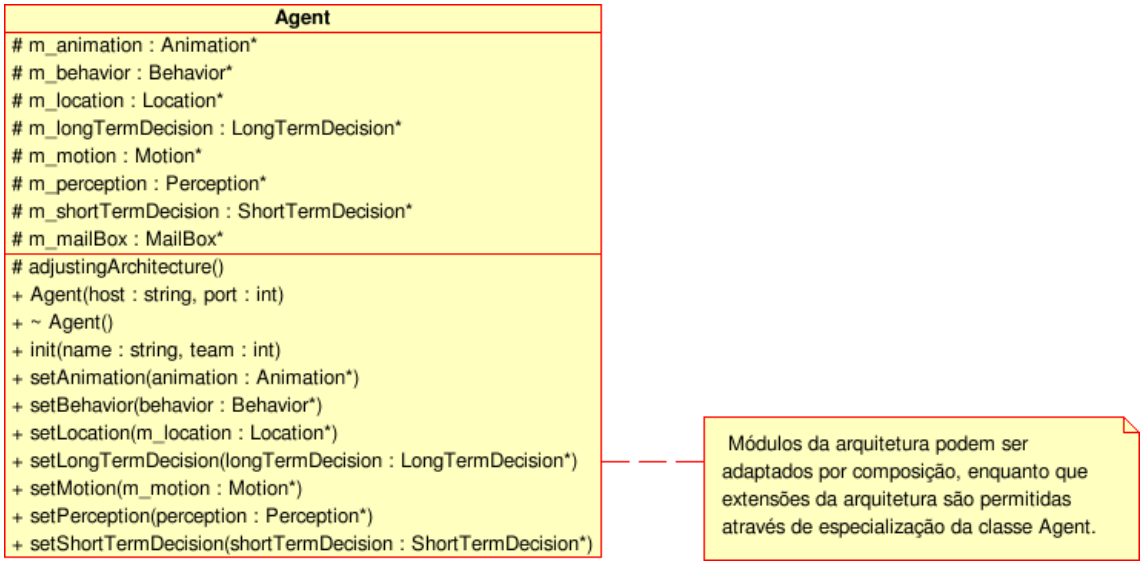

Figura A.2: Classe básica para a composição de um agente.

da composição de novos módulos, como pode ser visto na Figura A.2.

O módulo *Perception* tem como principal funcionalidade agrupar as informações sensoriais e entregar para os devidos módulos as informações necessárias. Ele é notificado pelo MailBox sempre que um nova mensagem chega.

Perception trata de forma diferente as mensagens síncronas das assíncronas. As mensagens assíncronas são enviadas contendo as informações do evento que disparou, desta forma, cada evento contém a sua mensagem assíncrona equivalente. Já para as mensagens síncronas, o que as disparam é o temporizador, mas neste caso o temporizador não dispara apenas uma mensagem contendo a informação necessária, mas sim, um conjunto de mensagens delimitado por mensagens de BEGIN e END. O módulo *Perception* é quem se encarrega de agrupar essas mensagens síncronas e só entregar para os outros módulos o bloco completo após o recebimento do END. Além disso, é possível fazer a seleção de quais mensagens sensorias devem ser passadas como percepção para cada módulo.

Uma outra funcionalidade do *Perception* é manter o histórico centralizado de percepções. Isto evita que cada módulo tente manter esse histórico a fim de realizar planejamento ou tomadas de decisões. As mensagens são armazenadas em uma fila e o tamanho dessa fila é configurável.

O módulo *Location* utiliza a representação interna do ambiente através de *WaypointSystem* e ele pode ser facilmente estendido para utilizar mais alguns outros tipos de representação, como campos potenciais(TOZOUR, 2004) ou mapas de influências(SCHWAB, 2004).

Location tem importante papel no processo cognitivo, servindo como uma representação interna do ambiente. O auxílio no planejamento de trajetória e identificação de elementos e regiões do ambiente são exemplos de funcionalidades fornecidas por esse módulo. Informações sobre os nós visitados e as características desses nós e suas vizinhanças auxiliam bastante no processo decisório do agente.

Long-Term Decision é o módulo utilizado para implementar a camada deliberativa do

agente. Sua implementação no *framework* é apenas estrutural, deixando a flexibilidade de extensão para a fase de projeto do agente. Sua estrutura facilita integração com outras arquiteturas baseadas em conhecimencimento como o SOAR(LAIRD; NEWELL; ROSENBLOOM, 1987). A idéia principal desse módulo é gerar planos que devem ser executados pelo *Short-Term Decision*.

Neste trabalho, o módulo *Long-Term Decision* foi implementado como uma elo de ligação entre o agente e o *proxy* de time. Assim, parte de suas atribuições são executadas pelo *proxy*, que delega papéis que o agente deve executar.

O módulo *Short-Term Decision* é o responsável pelo seqüenciamento das reações do agente, gerando um comportamento complexo de alto nível. Sua implementação no *framework* é apenas estrutural, deixando em aberto a forma como o seqüenciamento dos comportamentos é feito.

Neste trabalho, o módulo *Short-Term Decision* foi implementado através de máquinas de estados, formando uma coleção de máquinas especializadas em executar papéis específicos. Assim, para um dado papel designado por *Long-Term Decision* existe uma máquina de estado que o implementa.

O módulo *Motion* é utilizado principalmente para o planejamento de trajetória, implementando o algoritmo A* sobre a representação do ambiente em *Location*. Entretanto, para este trabalho, foi utilizado o *pahtfinder* fornecido pelo próprio jogo, fazendo do módulo *Motion* uma interface para a requisição de caminhos do próprio jogo.

O módulo *Animation* possui apenas sua implementação estrutural no *framework* e este trabalho não o utiliza porque o jogo UT2004 não fornece uma maneira direta de controlar a animação do personagem. As animações ficam a cargo do prórpio jogo.

O módulo *Behavior* atua diretamente com o ambiente, enviando os comandos básicos definidos pelo jogo. Ele possui uma lista de comportamentos na qual apenas um é ativado por vez. Estes comportamentos são ativados pelo módulo *Short-Term Decision*, e uma vez ativado, ele passa a responder por todas as percepções que chegem a *Behavior*. Apesar da possibilidade de implementação dos comportamentos de diferentes formas, o *framework* pré-disponibiliza alguns comportamentos implementados como máquinas de estados finitos. Esses comportamentos são utilizados como reações elementares pelo *Short-Term Decision*, que forma comportamentos complexos através de composição.

Os comportamentos implementados por padrão são específicos para a atuação nos jogos UT, UT2003 e UT2004. Uma importante observação quanto às máquinas de estados é que elas podem compartilhar estados com outras máquinas. Desta forma, um comportamento de "Tocaia"que necessite de um estado de "Aguardando Inimigo", pode compartilhar esse mesmo estado com um outro comportamento de "Protegendo Item", diminuindo o esforço com a implementação.

APÊNDICE B PERCEPÇÕES E AÇÕES COM GAMEBOTS

Este apêndice descreve as informações que estão contidas em cada tipo de mensagem trocada entro o jogo e o jogador virtual. Esta descrição está dividida em três partes. A primeira descreve as mensagens síncronas, a segunda as assíncronas e a terceira os comando possíveis de serem enviados para o jogo.

As mensagens síncronas são enviadas a um cliente em lotes dentro um intervalo pré-determinado. Tais mensagens incluem informações do que o *bot* enxerga e sobre o status do próprio *bot*. Todo lote de mensagens síncronas começam com uma mensagem do tipo BEG contendo um timestamp. Todas as mensagens subseqüentes farão parte do mesmo lote até que seja recebida uma mensagem do tipo END com o mesmo timestamp da mensagem BEG. Como todas essas mensagens são enviadas durante o mesmo instante de tempo de jogo, elas representam um único estado do jogo. Mensagens assíncronas, por sua vez, representam eventos que podem ocorrer a qualquer momento do jogo, como por exemplo sofrer dano ou se chocar com uma parede. É importante notar que apesar de poder ocorrer a qualquer momento do jogo uma mensagen assíncrona nunca irá ser enviada entre uma mensagem do tipo BEGIN e uma do tipo END de um lote e não há como saber o timestamp delas.

Mensagens síncronas:

- BEG - início de um lote de mensangens síncronas.
  - Time - timestamp do jogo.
- SLF - informação do estado do *bot*.
  - Id - id única, atribuída pelo jogo.
  - Rotation - para qual direção o player está voltado, em termos absolutos.
  - Location - localização absoluta.
  - Velocity - velocidade absoluta medida em unidades do jogo (unidades do jogo).
  - Name - nome que é utilizado pelo player.
  - Team - a qual time o player pertence. 255 significa que ele não está em time algum. 0-3 são os times vermelho, azul, verde e dourado respectivamente
  - Health - quanto de Health ainda resta para o player. Começa em 100 e varia de 0 a 200.

  - Weapon - qual arma o player está usando. Os nomes das armas incluem: ImpactHammer, Enforcer, Translocator, GESBioRifle, ShockRifle, PulseGun, Minigun2, UT_FlakCannon, UT_Eightball, WarheadLauncher.
  - CurrentAmmo - o quanto de munição resta na arma que está sendo usada no momento.
  - Armor - o quanto de proteção o *bot* possui. Começa em 0 e pode ir até 200.
- GAM - informação sobre o jogo.
  - PlayerScores - Lista dos scores de cada player da partida. Cada elemento dessa lista possui dois valores. O primeiro é o id do player e o segundo o valor do score.
  - TeamScores - funciona do mesmo jeito que a mensagen do tipo PlayerScores só que aplicada a times, onde os ids mostrados são os que identificam os times. Esse tipo de mensagen não é enviado quando a partida é no formato DeathMatch.
  - DomPoints - lista de pontos dominados por cada time para o formato de partida Domination.
  - HaveFlag - usado nas partidas do formato Capture The Flag (CTF) caso o *bot* esteja com a bandeira de outro time. O valor informado é o número do time dono da bandeira
  - EnemyHasFlag - usado nas partidas de CTF caso um *bot* de outro time esteja com a bandeira do time do seu *bot*
- PLR - Informações sobre outro player no jogo. Mostra somente informações sobre players que são visíves.
  - Id - id único atribuída pelo próprio jogo.
  - Rotation - para qual direção o player está voltado, em termos absolutos.
  - Location - localização absoluta.
  - Velocity - velocidade absoluta medida em unidades do jogo (unidades do jogo).
  - Team - a qual time o player pertence.
  - Reachable - tem valor true se é possíve alcançar esse player diretamente, o valor será falso caso contrário. Possíveis razões para este atributo ser falso é a existência de alguma espécie de obstáculo entre os dois players.
  - Weapon - que tipo de arma o player está segurando.
- NAV - informações sobre um *pathnode* do jogo. *Pathnodes* são objetos invisíveis colocados pelo mapa do jogo que definem caminhos para os *bot*s do jogo poderem seguir, gerando um grafo totalmente conexo que abrange quase todo mapa da partida.
  - Id - um id único atribuído a esse *pathnode* pelo próprio jogo.
  - Location - localização absoluta.

  - Reachable - tem valor true se o *bot* pode correr diretamente para este *path-node*.

- MOV - informações sobre objetos que podem se mover(portas, elevatores, etc).
  - Id - id único atribuído ao objeto móvel pelo próprio jogo.
  - Location - localização absoluta.
  - Reachable - tem valor true se o *bot* pode correr para este objeto.
  - DamageTrig - tem valor true se é necessário atirar no objeto para ativá-lo.
  - Class - classe do objeto móvel.
- DOM - ponto de dominação de uma partida de Domination.
  - Controller - qual time controla este ponto.
- FLG - informações sobre uma bandeira. (Somente em partidas de CTF).
  - Id - id único atribuído a essa bandeira pelo próprio jogo.
  - Location - localização absoluta.
  - Holder - id do *bot* que está carregando a bandeira.
  - Team - time ao qual a bandeira pertence.
  - Reachable - true se o *bot* pode correr diretamente para ela.
  - State - indica se a bandeira está com alguém (Held), caída no chão (Droped) ou está na base (Home)
- INV - informações sobre um objeto no chão que pode ser pego (item de inventário).
  - Id - id único atribuído a esse item de inventário pelo próprio jogo.
  - Location - localização absoluta.
  - Reachable - tem valor true se o *bot* pode correr diretamente para este objeto.
  - Class - uma string representando o tipo do objeto.
- END - indica final do lote de mensagens síncronas.
  - Time - timestamp do jogo.

Mensagens assíncronas:

- NFO - informações sobre o jogo logo após a conexão com o servidor ter sido estabelecida. O *bot* deve esperar esta mensagem antes de enviar uma mensagen do tipo INIT para o servidor.
  - Gametype - Qual o formato da partida (BotDeathMatchPlus, BotTeamGame, BotDomination).
  - Level - nome do mapa em que se está jogando.
  - TimeLimit - tempo máximo que o jogo irá durar.

  - FragLimit - número de mortes necessárias para vencer o jogo (somente para o formato BotDeathMatchPlus).
  - GoalTeamScore - número de pontos necessários para um time ganhar a partida (BotTeamGame, BotDomination).
  - MaxTeams - número máximo de times. Varia de 0 a (MaxTeams - 1) (BotTeamGame, BotDomination).
  - MaxTeamSize - número máximo de players por time (BotTeamGame, BotDomination)
- AIN - enviada pelo servidor quando o *bot* adquire um novo item para o inventário.
  - Id - id único atribuído a este item pelo próprio jogo. Id baseado em uma string que descreve o tipo do item.
  - Class - uma string representando o tipo do item.
- VMS - mensagem recebida do canal de chat global.
  - String - mensagem enviada por outro player no jogo através do canal global.
- VMT - mensagem recebida do canal de chat privado.
  - String - mensagem enviada por um player do mesmo time através do canal privado do time.
- VMG - mensagem padrão do jogo recebida de outro player.
  - Sender - id único do player que enviou a mensagem.
  - Type - tipo da mensagem (comando, provocação).
  - Id - id da mensagem.
- ZCF - pés do *bot* mudaram de uma zona artificial do jogo para outra. (se entrou na água ou na lava)
  - Id - id único da zona em que os pés estão agora.
- ZCH - a cabeça do *bot* mudou de uma zona artificial para outra.
  - Id - id único da zona em que a cabeça está agora.
- ZCB - *bot* mudou de uma zona artificial para outra.
  - Id - id único da zona em que o *bot* está agora.
- CWP - o *bot* trocou de arma (através de comando ou porque a munição da arma terminou).
  - Id - id único da nova arma, baseado no nome da arma.
  - Class - uma string representando o tipo da arma
- WAL - houve colisão com uma parede.
  - Id - id único da parede onde colidiu.
  - Normal - normal do ângulo que o *bot* colidiu.

  - Location - localização absoluta do *bot* no momento da colisão.
- FAL - *bot* atingiu a beirada de algum lugar alto. Se o *bot* estava caminhando ele não terá caído, caso esteja correndo já estará caíndo no recebimento desta mensagem.
  - Fell - True se caiu, falso se parou na beirada.
  - Location - localização absoluta do *bot*
- BMP - se chocou com outro ator.
  - Id - id único do ator (atores são outros players ou qualquer objeto que possa bloquear seu caminho).
  - Location - localização do objeto com o qual o *bot* se chocou.
- HRP - o *bot* escutou alguém pegar algum objeto do chão.
  - Player - id único do player que pegou o objeto.
- HRN - o *bot* escutou um barulho (um outro player andando ou atirando, uma bala acertando o chão ou um elevador se movendo).
  - Source - id único do ator que fez o barulho.
- SEE - o *bot* viu um outro player. Mensagem gerada pelo jogo entre 1 e 2 vezes a cada segundo quando outro player está visível. Útil quando o delay entre as mensagens síncronas é muito grande É possível que esteja depreciado.
  - Id - id único do player, atribuído pelo jogo.
  - Rotation - para qual direção o player está voltado em termos absolutos.
  - Location - localização absoluta do player.
  - Velocity - localização absoluta em unidades do jogo.
  - Team - em qual time o player está.
  - Reachable - true se é possível correr em direção a este player.
  - Weapon - qual arma o player está usando.
- PRJ - projétil vindo em sua direção.
  - Time - tempo estimado para o impacto.
  - Direction - valor de rotação de onde o projétil está vindo.
- KIL - algum outro player morreu.
  - Id - id único do player.
  - Killer - id único do player que o matou, caso exista.
  - DamageType - string descrevendo o tipo de dano que o matou.
- DIE - este *bot* morreu.
  - Killer - id único do player que o matou, caso exista.
  - DamageType - string descrevendo o tipo de dano que o matou.

- DAM - o *bot* sofreu dano.
  - Damage - quantidade de dano sofrido.
  - DamageType - string descrevendo o tipo de dano.
- HIT - feriu outro player.
  - Id - id único do player ferido.
  - Damage - quantidade de dano provocado.
  - DamageType - string descrevendo o tipo de dano.
- PTH - lista de *pathnodes* em resposta a uma mensagem GETPATH do cliente.
  - Id - id da mensagem idêntica à enviada pelo cliente, permitindo que este possa reconhecer a resposta da sua mensagem.
  - Multiple pathnodes: lista de nós para se chegar no destino desejado na ordem em que precisam ser percorridos.
- RCH - resultado booleano em resposta a um comando do tipo CHECKREACH.
  - Id - id da mensagem idêntica à enviada pelo cliente, permitindo que este possa reconhecer a resposta da sua mensagem.
  - Reachable - true se o *bot* pode se movimentar na direção informada, falso caso contrário.
  - From - localização do *bot* no momento do check.
- FIN - enviada quando a partida termina.

Comandos:

- INIT - mensagem enviada para que seja instanciado um personagem no jogo, deve ser enviada antes de poder executar qualquer ação e somente após receber uma mensagem do tipo NFO do servidor.
  - Name - Nome a ser usado pelo *bot*.
  - Team - Diz o time ao qual o *bot* pertencerá.
- SETWALK - diz se o *bot* irá se movimentar andando ou correndo.
  - Walk - Se tiver valor True o *bot* irá caminhar, caso contrário irá correr.
- STOP - para qualquer movimentação que esteja executando.
- JUMP - faz o *bot* pular.
- RUNTO - se movimenta para uma determinada localização ou alvo.
  - Target - id único do alvo para o qual deseja se movimentar. Só irá funcionar se o alvo estiver visível, caso contrário o *bot* não fará nada.
  - Location - As três coordenadas da localização para onde o *bot* deve se movimentar.

- STRAFE - semelhante ao RUNTO, mas o *bot* irá se movimentar em uma direção enquanto está olhando para outro ponto.
  - Location - As três coordenadas da localização para onde o *bot* deve se movimentar.
  - Target - id único do alvo para o qual deseja se movimentar. Só irá funcionar se o alvo estiver visível.
- TURNTO - faz o *bot* virar ou para a direção de um alvo ou para uma localização.
  - Target - id único do alvo para o qual deseja se movimentar. Só irá funcionar se o alvo estiver visível.
  - Rotation - Valor de rotação que se deseja usar. Três coordenadas em unidades do jogo em valores absolutos, separadas por espaço ou vírgula (2pi = 65535 unidades do jogo). Esse argumento somente é usado se nenhum alvo for indicado.
  - Location - As três coordenadas da localização para onde o *bot* deve se virar. Somente usada se não forem passados nem alvo nem valor de rotação.
- ROTATE - rotaciona uma quantidade específica.
  - Amount - quantidade a ser rotacionada em unidades do jogo. Pode ser negativo para rotacionar em sentido anti-horário.
  - Axis - se fornecido o eixo vertical, a rotação será em Pitch. Qualquer outro valor a rotação será em Yaw.
- SHOOT - faz com que o *bot* comece a atirar.
  - Location - As três coordenadas da localização para onde o *bot* deve atirar.
  - Target - id único do alvo no qual o *bot* deve atirar. Se o alvo estiver visível o servidor irá providendicar correção da mira e irá fazer com que o *bot* continue mirando no alvo. Caso não esteja visível o *bot* somente irá ficar atirando na localização informada. Mesmo informando o alvo ainda é necessário informar a localização.
  - Alt - Faz com que o *bot* use o disparo alternativo da arma.
- CHANGEWEAPON - o *bot* troca de arma.
  - Id - id único da arma que o *bot* deve usar. Se for usado o valor Best será escolhida a melhor arma que ainda tenha munição.
- STOPSHOOT - faz o *bot* parar de atirar.
- CHECKREACH - checa se é possíve se movimentar diretamente para algum lugar ou algo sem ser obstruído.
  - Target - id único do alvo para o qual se deseja fazer o check. É necessário que ele esteja visível.
  - Location - As três coordenadas da localização para onde o *bot* deve atirar. Somente usado se nenhum alvo for informado.

  - Id - id da mensagem criada pelo *bot* para ser retransmitida em resposta pelo servidor, assim é possível identificar a resposta da mensagem enviada pelo *bot*.
  - From - localização do *bot* no momento do check.
- GETPATH - obtem um caminho para uma localização específica. É retornada uma lista ordenada de *pathnodes*.
  - Location - As três coordenadas da localização para onde o *bot* deve ir.
  - Id - id da mensagem criada pelo *bot* para ser retransmitida em resposta pelo servidor, assim é possível identificar a resposta da mensagem enviada pelo *bot*.
- MESSAGE - envia uma mensagem para todos ou somente para os integrantes do seu time.
  - String - string a ser enviada.
  - Global - Se o valor for true, envia a mensagem para todos, caso contrário somente para o time.
- PING - envia um PING para o servidor que irá responder com um PONG.

APÊNDICE C RESULTADOS DAS PARTIDAS

Neste apêndice, são apresentados os resultados das partidas de *Capturar a Bandeira*. Aqui, são detalhadas as pontuações obtidas por cada jogador juntamente com a pontuação do time, sendo muitas vezes possível observar o tipo de estratégias predominantes no combate. Por exemplo, um time que possui um jogador que se destaca muito dos demais na pontuação individual é um indicativo de que esse jogador assumia uma posição de atirador de elite, o que é bastante visível nas batalhas contra times humanos. Também é possível identificar quando um time possuía jogadores específicos protegendo a base, pois esses jogadores normalmente obtinham as menores pontuações individuais, já que estavam a maior parte do tempo fora das zonas de combate.

A Tabela C.1 descreve os resultados das partidas entre o *TWProxy-Team* e o *UTBots-Average*. É possível observar que os jogadores do *TWProxy-Team* tiveram desempenho muito melhor que os jogadores do *UTBots-Average*, mesmo na partida em que o *UTBots-Average* conseguiu marcar 1 ponto.

Na Tabela C.2 estão os resultados das partidas entre o *TWProxy-Team* e o *UTBots-Experienced*. Apesar do *UTBots-Experienced* possuir um desempenho individual melhor que o *UTBots-Average*, ele não conseguiu pontuar sobre o *TWProxy-Team*, que o venceu com facilidade.

O resultado das partidas entre *Machinetta-Team* e *UTBots-Average*, apresentado na Tabela C.3, mostra que os jogadores do *Machinetta-Team* tiveram desempenho individual bem superior ao *UTBots-Average*, entretanto, o desempenho do time foi muito baixo comparado ao *TWProxy-Team* na mesma situação.

A Tabela C.4 mostra como foram disputados os combates entre o *TWProxy-Team* e o *Machinetta-Team*. Os dois times foram compostos pelos mesmos jogadores, por isso, o desempenho individual dos jogadores são próximos. Entretanto, a diferença no desempenho dos times evidencia as vantagens do *TWProxy* frente ao *Machinetta*.

Os resultados apresentados na Tabela C.5 mostram partidas bastante heterogêneas entre o *TWProxy-Team* e o time de humanos. Nessas partidas, os humanos freqüentemente variavam suas estratégias de jogo e é possível notar que na maioria das vitórias do time de humanos existe um jogador com pontuação individual bem acima dos demais. O motivo dessa diferença está no papel de atirador de elite que emergia nos times com humanos. Esses atiradores de elite tinham sempre uma pontuação superior aos demais por atingir um grande número de inimigos sem ser visto, dando cobertura ao resto do time que efetuava a captura da bandeira.

Tabela C.1: Resultado das partidas de *Capture The Flag* entre agentes usando *TWProxy* e bots do jogo no nível *Average*.

TWProxy-Team						**X**	**UTBots-Average**					
Pontuação individual					*Pontos do Time*	X	*Pontos do Time*	*Pontuação individual*				
25	35	37	47	61	**1**	X	**0**	39	36	25	14	6
12	18	33	59	66	**1**	X	**0**	22	21	19	14	12
32	40	41	44	57	**2**	X	**0**	39	21	20	19	17
19	24	48	64	71	**1**	X	**0**	30	27	26	25	17
9	28	45	47	74	**1**	X	**0**	51	36	21	19	16
34	44	48	66	74	**5**	X	**0**	28	20	12	10	7
14	43	47	50	64	**2**	X	**0**	28	21	12	11	8
12	46	47	53	64	**2**	X	**0**	33	17	14	14	9
35	43	49	64	67	**4**	X	**0**	25	23	18	10	9
29	35	50	60	77	**3**	X	**0**	23	21	21	20	7
28	43	53	57	67	**3**	X	**0**	35	29	28	12	10
39	50	53	59	85	**4**	X	**0**	27	24	20	19	14
10	33	38	53	76	**1**	X	**0**	25	19	17	17	16
42	56	57	59	72	**5**	X	**0**	30	16	10	9	6
25	37	49	51	102	**3**	X	**0**	31	27	23	16	3
4	48	60	74	85	**4**	X	**0**	37	33	13	7	6
43	45	47	51	51	**4**	X	**0**	39	16	16	15	2
24	33	42	57	66	**2**	X	**0**	27	26	20	14	9
22	45	46	50	56	**2**	X	**0**	30	26	20	17	10
19	33	50	53	74	**2**	X	**0**	31	23	23	13	13
35	41	44	52	87	**3**	X	**0**	32	25	20	20	10
28	42	57	59	77	**1**	X	**0**	75	28	24	19	9
17	43	45	58	73	**5**	X	**0**	34	11	9	9	6
18	24	43	58	66	**0**	X	**0**	31	26	25	24	14
43	44	49	54	72	**4**	X	**0**	24	23	19	16	13
24	33	38	57	76	**3**	X	**0**	37	21	18	15	13
34	35	41	55	61	**3**	X	**0**	35	27	14	13	3
20	26	29	43	63	**2**	X	**1**	26	21	14	10	7
25	25	26	33	43	**1**	X	**0**	33	18	12	11	6
5	34	37	54	85	**0**	X	**0**	45	30	20	20	13
20	28	41	50	57	**1**	X	**0**	38	33	18	16	9

Tabela C.2: Resultado das partidas de *Capture The Flag* entre agentes usando *TWProxy* e bots do jogo no nível *Experienced*.

TWProxy-Team						**X**	**UTBots-Experienced**					
Pontuação individual					*Pontos do Time*	X	*Pontos do Time*	*Pontuação individual*				
24	42	47	62	63	**2**	X	**0**	47	20	15	12	12
38	45	50	52	61	**2**	X	**0**	33	30	26	24	17
17	31	56	65	80	**3**	X	**0**	28	24	18	18	14
15	37	43	50	60	**1**	X	**0**	49	37	26	11	11
19	20	27	69	75	**1**	X	**0**	40	39	32	28	3
31	34	43	63	95	**3**	X	**0**	29	25	24	22	12
15	37	38	45	46	**0**	X	**0**	34	31	29	21	21
27	45	54	54	54	**2**	X	**0**	35	25	24	17	12
28	49	50	57	64	**0**	X	**0**	49	46	29	16	12
19	43	48	52	53	**1**	X	**0**	36	27	26	26	11
29	31	48	51	61	**1**	X	**0**	35	33	21	18	15
16	32	39	76	82	**3**	X	**0**	30	24	21	20	15
23	39	49	57	73	**0**	X	**0**	63	39	21	13	8
27	44	51	57	74	**4**	X	**0**	30	29	19	15	8
23	45	50	55	79	**2**	X	**0**	43	32	20	18	15
31	32	39	62	82	**1**	X	**0**	36	29	22	18	15
32	39	43	52	55	**1**	X	**0**	35	25	25	20	17
31	39	52	63	76	**0**	X	**0**	64	46	42	33	12
38	54	56	60	74	**5**	X	**0**	23	20	17	17	14
29	30	38	50	63	**1**	X	**0**	35	25	23	18	11
29	33	45	54	67	**1**	X	**0**	29	25	22	21	18
29	37	44	57	67	**1**	X	**0**	39	28	21	18	11
6	25	30	43	50	**0**	X	**0**	26	3	3	3	1
33	37	44	47	48	**1**	X	**0**	28	26	21	18	10
27	36	45	49	54	**0**	X	**0**	40	31	24	19	16
13	49	49	52	64	**1**	X	**0**	33	27	20	19	18
31	33	33	43	59	**0**	X	**0**	49	30	26	23	17
21	26	36	44	64	**1**	X	**0**	30	22	20	15	7
29	30	32	47	61	**2**	X	**0**	15	14	14	13	10
32	40	47	49	68	**1**	X	**0**	32	22	21	20	18

Tabela C.3: Resultado das partidas de *Capture The Flag* entre agentes usando *Machinetta* e bots do jogo no nível *Average*.

Machinetta-Team						**X**	**UTBots-Average**					
Pontuação individual					*Pontos do Time*	X	*Pontos do Time*	*Pontuação individual*				
45	55	58	62	69	**0**	X	**0**	26	23	18	8	6
6	8	34	48	55	**0**	X	**0**	25	11	11	7	6
3	8	56	57	63	**0**	X	**0**	30	16	14	13	11
1	56	69	74	74	**0**	X	**0**	32	20	15	15	13
46	48	49	51	63	**0**	X	**0**	31	16	16	15	13
14	23	23	25	31	**0**	X	**0**	8	4	4	0	0
8	7	18	72	74	**0**	X	**0**	19	18	17	13	11
6	8	32	44	62	**1**	X	**0**	16	11	9	4	1
0	2	54	73	86	**0**	X	**0**	35	22	13	10	10
7	11	57	65	72	**0**	X	**0**	23	23	22	17	15
8	10	10	19	84	**0**	X	**0**	20	10	7	6	5
14	23	33	36	41	**1**	X	**0**	6	5	2	1	0
11	23	30	31	76	**3**	X	**1**	31	29	19	11	11
4	14	21	41	67	**1**	X	**0**	11	10	8	7	2
6	6	53	61	68	**0**	X	**0**	29	23	17	13	12
8	11	13	69	70	**0**	X	**0**	23	21	13	8	6
8	18	18	26	75	**0**	X	**0**	20	12	11	5	4
17	26	28	29	67	**0**	X	**0**	19	12	11	7	4
13	16	22	40	42	**0**	X	**0**	13	10	9	7	7
0	7	63	72	80	**0**	X	**0**	30	20	15	13	12
6	10	59	65	74	**0**	X	**0**	47	32	31	26	14
8	21	37	61	63	**0**	X	**0**	27	25	22	15	4
12	25	51	59	72	**1**	X	**0**	34	17	13	11	11
2	3	56	69	71	**0**	X	**0**	34	23	21	17	15
4	9	36	39	78	**1**	X	**0**	15	10	7	6	5
13	14	18	27	80	**0**	X	**0**	34	19	18	15	15
9	12	12	56	81	**0**	X	**0**	18	14	14	12	12
13	17	25	26	31	**0**	X	**0**	9	6	1	1	1
7	8	59	60	75	**0**	X	**0**	37	19	18	14	11
5	58	58	75	75	**0**	X	**0**	43	21	17	13	12

Tabela C.4: Resultado das partidas de *Capture The Flag* entre agentes usando *TWProxy* e agentes usando *Machinetta*.

TWProxy-Team						**X**	**Machinetta-Team**					
Pontuação individual					*Pontos do Time*	X	*Pontos do Time*	*Pontuação individual*				
20	33	50	51	55	**1**	X	**0**	46	32	21	15	10
22	45	53	60	72	**0**	X	**0**	52	45	41	39	21
27	52	54	67	70	**2**	X	**0**	58	57	25	20	17
4	23	74	92	115	**0**	X	**0**	59	54	44	38	15
3	19	41	71	78	**0**	X	**0**	75	71	47	14	5
0	35	49	50	52	**0**	X	**0**	69	42	21	15	14
4	40	67	68	75	**1**	X	**0**	64	63	55	14	10
2	34	72	101	111	**1**	X	**0**	49	40	33	28	19
26	30	31	48	53	**0**	X	**0**	78	62	26	14	8
7	48	64	71	80	**0**	X	**0**	97	61	37	15	14
0	25	44	48	55	**0**	X	**0**	47	37	31	24	22
26	32	38	55	58	**0**	X	**0**	79	46	27	18	13
6	26	31	47	52	**0**	X	**0**	76	34	33	24	19
23	28	42	66	67	**1**	X	**0**	50	49	32	23	20
5	12	50	66	80	**0**	X	**0**	62	48	31	18	13
18	30	37	60	69	**0**	X	**0**	50	44	40	27	11
29	53	54	64	74	**2**	X	**0**	57	44	45	27	21
17	43	48	52	57	**0**	X	**0**	67	37	31	25	23
46	49	52	62	83	**2**	X	**0**	61	50	45	28	20
9	23	30	41	49	**1**	X	**0**	71	26	18	18	14
0	16	29	47	63	**0**	X	**0**	72	27	27	23	16
1	11	56	56	78	**1**	X	**0**	68	57	21	19	14
3	11	54	61	63	**0**	X	**0**	54	53	22	13	11
2	35	38	56	74	**0**	X	**0**	73	59	23	18	12
17	50	74	77	97	**3**	X	**0**	58	51	35	32	12
13	56	61	70	82	**1**	X	**0**	60	56	43	31	24
0	1	13	36	91	**1**	X	**0**	30	22	15	13	10
8	11	19	23	29	**0**	X	**0**	34	26	26	25	16
10	10	27	37	52	**0**	X	**0**	59	22	22	20	16
11	12	44	47	86	**1**	X	**0**	79	63	14	14	12

Tabela C.5: Resultado das partidas de *Capture The Flag* entre agentes usando *TWProxy* e jogadores humanos.

TWProxy-Team					**X**	**Human-Team**				
Pontuação individual				*Pontos do Time*	X	*Pontos do Time*	*Pontuação individual*			
5	16	28	28	**0**	X	**1**	58	43	17	14
12	22	24	26	**0**	X	**0**	77	37	23	14
19	24	28	29	**0**	X	**0**	81	60	35	30
13	21	25	29	**0**	X	**3**	76	70	36	27
19	21	27	32	**0**	X	**0**	36	24	21	17
33	39	41	53	**0**	X	**0**	48	43	22	16
22	32	43	46	**0**	X	**2**	55	51	48	16
14	27	34	49	**0**	X	**0**	31	23	17	11
7	23	26	45	**0**	X	**0**	65	32	24	17
16	19	21	25	**0**	X	**0**	88	71	26	13
21	23	27	33	**0**	X	**0**	38	34	28	24
18	23	32	44	**1**	X	**0**	72	26	16	6
15	19	21	23	**0**	X	**0**	109	28	23	13
15	23	24	24	**0**	X	**0**	88	52	25	20
9	24	25	40	**0**	X	**1**	99	34	25	13
8	18	20	55	**1**	X	**0**	32	25	21	19
10	21	29	30	**0**	X	**1**	66	34	30	22
19	19	30	43	**1**	X	**0**	51	17	12	2
23	32	34	40	**0**	X	**0**	72	34	31	10
21	26	26	33	**0**	X	**0**	67	30	28	26
8	19	32	46	**0**	X	**0**	65	45	28	25
16	20	23	23	**0**	X	**1**	116	52	26	26
17	19	44	47	**0**	X	**0**	39	35	34	14
22	23	37	51	**1**	X	**0**	50	43	23	15
25	36	40	52	**1**	X	**1**	96	65	29	24
19	34	34	37	**1**	X	**0**	98	35	29	19
15	16	17	20	**0**	X	**0**	66	44	39	11
25	27	32	33	**0**	X	**0**	64	57	44	19
28	37	44	48	**1**	X	**0**	44	39	25	20
10	30	34	39	**0**	X	**4**	68	63	46	31

APÊNDICE D ALGORITMO HÚNGARO

O método húngaro é um algoritmo de otimização combinacional que resolve o problema de atribuição em tempo polinomial. Ele foi desenvolvido e publicado em 1955 por Harold Kuhn (KUHN, 1955) e revisado em 1957 por James Munkres (MUNKRES, 1957).

Neste apêndice é apresentada uma implementação em C++ do algoritmo húngaro utilizado no *TWProxy*. A classe *MunkresAllocation*, que extende *RoleAllocation*, tem a sua interface apresentada na Listagem D.1 e seu fonte na Listagem D.2.

Listagem D.1: MunkresAllocation.h

```
/*************************************************************************
 *   Copyright (C) 2008 by Ivan Medeiros Monteiro   *
 *   fehler@gmail.com   *
 *                                                                       *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU Library General Public License as       *
 *   published by the Free Software Foundation; either version 2 of the    *
 *   License, or (at your option) any later version.                       *
 *                                                                         *
 *   This program is distributed in the hope that it will be useful,       *
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of        *
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the         *
 *   GNU General Public License for more details.                          *
 *                                                                         *
 *   You should have received a copy of the GNU Library General Public     *
 *   License along with this program; if not, write to the                 *
 *   Free Software Foundation, Inc.,                                       *
 *   59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.             *
 *************************************************************************/
#ifndef MUNKRESALLOCATION_H
#define MUNKRESALLOCATION_H

#include "RoleAllocation.h"

#include <vector>
#include <string>
/**
    @author Ivan Medeiros Monteiro <fehler@gmail.com>
*/

using std::vector;
using std::string;


class MunkresAllocation : public RoleAllocation{
    private:
        unsigned int m_matSize;
        vector<bool> m_coveredRows;
        vector<bool> m_coveredCols;
        vector<vector<float> > m_original;
        vector<vector<float> > m_cost;
        vector<vector<char> > m_mask;
```

```
        vector<bool> m_hasTask;

        unsigned int m_currentZeroRow;
        unsigned int m_currentZeroCol;

        unsigned int m_nextStep;

        float m_maxValue;

        void step1();
        void step2();
        void step3();
        void step4();
        void step5();
        void step6();

        bool findZero(unsigned int &row, unsigned int &col);
        bool findStarInRow(unsigned int row, unsigned int &col);
        bool findStarInCol(unsigned int &row, unsigned int col);
        void clearCover();
        unsigned int findPrimeInRow(unsigned int row);
        void erasePrimes();
        void convertPath(vector<unsigned int> &path1, vector<unsigned int> &path2);
        float findSmallest();

        void fillRowsAndCols();
        void copyCap(unsigned int src, unsigned int dst);

    public:
        MunkresAllocation(unsigned int nAgents, unsigned int nRoles);

        ~MunkresAllocation();



        void setCapability(string rapId, string roleId, float cap);
        vector<pair<string, string> > &allocate();
};

#endif
```

Listagem D.2: MunkresAllocation.cpp

```
/*************************************************************************
 *   Copyright (C) 2008 by Ivan Medeiros Monteiro   *
 *   fehler@gmail.com   *
 *                                                                       *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU Library General Public License as       *
 *   published by the Free Software Foundation; either version 2 of the    *
 *   License, or (at your option) any later version.                       *
 *                                                                         *
 *   This program is distributed in the hope that it will be useful,       *
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of        *
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the         *
 *   GNU General Public License for more details.                          *
 *                                                                         *
 *   You should have received a copy of the GNU Library General Public     *
 *   License along with this program; if not, write to the                 *
 *   Free Software Foundation, Inc.,                                       *
 *   59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.             *
 *************************************************************************/
#include "MunkresAllocation.h"

#include <iostream>

MunkresAllocation::MunkresAllocation(unsigned int nAgents, unsigned int nRoles)
  : RoleAllocation(nAgents, nRoles){
    vector<float> vf;
    vector<char> vc;

    m_matSize = nAgents > nRoles ? nAgents : nRoles;
```

```
    m_nextStep = 1;

    m_maxValue = 0.0f;

    for(unsigned int i=0; i<m_matSize; i++){
        vf.push_back(0.0001f);
        vc.push_back(0);
    }


    for (unsigned int i=0; i<m_matSize; i++){
        m_coveredRows.push_back(false);
        m_coveredCols.push_back(false);
        m_hasTask.push_back(false);
        m_original.push_back(vf);
        m_cost.push_back(vf);
        m_mask.push_back(vc);
    }

}


MunkresAllocation::~MunkresAllocation(){
}

void MunkresAllocation::fillRowsAndCols(){
    char buffer[32];
    if(m_nAgents < m_matSize){
        unsigned int n = m_matSize-m_nAgents;
        for(unsigned int i=0; i<n; i++){
            sprintf(buffer, "unknown_%d", i);
            addAgent(buffer);
        }
    }
    else if(m_nRoles < m_matSize){
        unsigned int n = m_matSize-m_nRoles;
        if(m_flexibleRoles.size()){
            for(unsigned int i=0; i<n; i++){
                string roleId = m_flexibleRoles[rand()%m_flexibleRoles.size()];
                copyCap(m_roleIndex[roleId], m_nRoles+i);
                sprintf(buffer, "_%d", i);
                addRole(roleId + buffer);
            }
        }
        else{
            for(unsigned int i=0; i<n; i++){
                sprintf(buffer, "unknown_%d", i);
                addRole(buffer);
            }
        }
    }

}

void MunkresAllocation::copyCap(unsigned int src, unsigned int dst){
    for(unsigned int i=0; i<m_matSize; i++){
        m_original[i][dst] = m_original[i][src];
        m_cost[i][dst] = m_cost[i][src];
    }
}


void MunkresAllocation::setCapability(string rapId, string roleId, float cap){
    unsigned int agentIndex = m_agentIndex[rapId];
    unsigned int roleIndex = m_roleIndex[roleId];

    m_original[agentIndex][roleIndex] = cap;

    if(cap > m_maxValue) m_maxValue = cap;
}


vector<pair<string, string> > &MunkresAllocation::allocate(){
```

```
    fillRowsAndCols();
    m_allocated.clear();

    for(unsigned int i=0; i<m_matSize; i++){
        for(unsigned int j=0; j<m_matSize; j++){
            m_cost[i][j] = m_maxValue - m_original[i][j];
        }
    }


    while(m_nextStep!=7){
        switch(m_nextStep){
            case 1:
                step1();
                break;
            case 2:
                step2();
                break;
            case 3:
                step3();
                break;
            case 4:
                step4();
                break;
            case 5:
                step5();
                break;
            case 6:
                step6();
                break;
            default:
                break;
        }


    }

    for(unsigned int i=0; i<m_matSize; i++){
        for(unsigned int j=0; j<m_matSize; j++){
            std::cout << m_mask[i][j] + 0 << " " ;
            if(m_mask[i][j] == 1 && !m_hasTask[i]) {
                m_hasTask[i] = true;
                m_allocated.push_back(pair<string,string>(m_agentIds[i],
                        m_roleIds[j].substr(0, m_roleIds[j].rfind('_')) ));
            }
        }
        std::cout << std::endl;
    }

    return m_allocated;
}

void MunkresAllocation::step1(){
    float minVal;

    for(unsigned int i=0 ; i<m_matSize; i++){
        minVal = m_cost[i][0];
        for(unsigned int j=1; j<m_matSize; j++){
            if(m_cost[i][j] < minVal) minVal = m_cost[i][j];
        }
        for(unsigned int j=0; j<m_matSize; j++){
            m_cost[i][j] -= minVal;
        }
    }

    m_nextStep =2;
}

void MunkresAllocation::step2(){
    for(unsigned int i=0 ; i<m_matSize; i++){
        for(unsigned int j=0; j<m_matSize; j++){
            if(m_cost[i][j] < 0.00001f && !m_coveredCols[j] && !m_coveredRows[i]){
```

```
                m_mask[i][j] = 1;
                m_coveredRows[i] = true;
                m_coveredCols[j] = true;
            }
        }
    }

    for(unsigned int i=0 ; i<m_matSize; i++){
        m_coveredCols[i] = false;
        m_coveredRows[i] = false;
    }

    m_nextStep=3;
}

void MunkresAllocation::step3(){
    unsigned int count=0;

    for(unsigned int i=0 ; i<m_matSize; i++){
        for(unsigned int j=0; j<m_matSize; j++){
            if(m_mask[i][j]==1){
                m_coveredCols[j]=true;
            }
        }
    }

    for(unsigned int j=0; j<m_matSize; j++){
        if(m_coveredCols[j])count++ ;
    }

    if(count >= m_matSize) m_nextStep = 7;
    else m_nextStep =4;
}

void MunkresAllocation::step4(){
    bool done=false;
    unsigned int row, col;
    m_nextStep=5;

    while(!done){
        if(!findZero(row, col)){
            done = true;
            m_nextStep = 6;
        }
        else {
            m_mask[row][col] = 2;
            if(findStarInRow(row,col)){
                m_coveredRows[row] = true;
                m_coveredCols[col] = false;
            }
            else{
                done=true;
                m_nextStep = 5;
                m_currentZeroRow = row;
                m_currentZeroCol = col;
            }
        }
    }
}

void MunkresAllocation::step5(){
    vector<unsigned int> path1;
    vector<unsigned int> path2;
    unsigned int count=0;
    unsigned int row;

    path1.push_back(m_currentZeroRow);
    path2.push_back(m_currentZeroCol);

    while(true){
        if(findStarInCol(row, path2[count])){
            count++;
            path1.push_back(row);
```

```
            path2.push_back(path2[count-1]);
        }
        else break;

        count++;
        path1.push_back(path1[count-1]);
        path2.push_back(findPrimeInRow(path1[count-1]));
    }
    convertPath(path1, path2);
    clearCover();
    erasePrimes();
    m_nextStep = 3;
}


void MunkresAllocation::step6(){
    float minVal = findSmallest();

    for(unsigned int i=0 ; i<m_matSize; i++){
        for(unsigned int j=0; j<m_matSize; j++){
            if(m_coveredRows[i]){
                m_cost[i][j] += minVal;
            }

            if(!m_coveredCols[j]){
                m_cost[i][j] -= minVal;
            }
        }
    }
    m_nextStep=4;
}



bool MunkresAllocation::findZero(unsigned int &row, unsigned int &col){
    for(unsigned int i=0 ; i<m_matSize; i++){
        for(unsigned int j=0; j<m_matSize; j++){
            if(m_cost[i][j] < 0.00001f && !m_coveredRows[i] && !m_coveredCols[j]){
                row = i;
                col = j;
                return true;
            }
        }
    }

    return false;
}


bool MunkresAllocation::findStarInRow(unsigned int row, unsigned int &col){
    for(unsigned int j=0 ; j<m_matSize; j++){
        if(m_mask[row][j] == 1){
            col = j;
            return true;
        }
    }
    return false;
}


bool MunkresAllocation::findStarInCol(unsigned int &row, unsigned int col){
    for(unsigned int i=0 ; i<m_matSize; i++){
        if(m_mask[i][col] == 1){
            row = i;
            return true;
        }
    }
    return false;
}


void MunkresAllocation::clearCover(){
    for(unsigned int i=0 ; i<m_matSize; i++){
```

```
        m_coveredRows[i]=false;
        m_coveredCols[i]=false;
    }
}


unsigned int MunkresAllocation::findPrimeInRow(unsigned int row){
    for(unsigned int j=0 ; j<m_matSize; j++){
        if(m_mask[row][j] == 2){
            return j;
        }
    }
    throw "it should not happen.";
}


void MunkresAllocation::erasePrimes(){
    for(unsigned int i=0 ; i<m_matSize; i++){
        for(unsigned int j=0; j<m_matSize; j++){
            if(m_mask[i][j] == 2) m_mask[i][j] = 0;
        }
    }
}


void MunkresAllocation::convertPath(vector<unsigned int> &path1,
                                    vector<unsigned int> &path2){
    for(unsigned int i=0; i < path1.size(); i++){
        m_mask[path1[i]][path2[i]] = m_mask[path1[i]][path2[i]] == 1 ?  0 : 1;
    }
}


float MunkresAllocation::findSmallest(){
    float minVal = 1e32;

    for(unsigned int i=0 ; i<m_matSize; i++){
        for(unsigned int j=0; j<m_matSize; j++){
            if(!m_coveredRows[i] && !m_coveredCols[j] && minVal > m_cost[i][j]){
                minVal = m_cost[i][j];
            }
        }
    }

    return minVal;
}
```