3 ASYNC – Um Framework para Construção de Agentes Inteligentes

Este capítulo apresenta o Agent's SYNergistic Cooperation (ASYNC) *framework*. O ASYNC (Sardinha et al., 2003a) é um *framework* orientado a objetos que permite construir agentes de software em ambientes distribuídos. O *framework* proposto foi concebido através de um desenvolvimento *bottom-up*. Três aplicações baseadas em agentes foram desenvolvidas para identificar o código re-utilizável: (i) Uma ferramenta (Sardinha, 2001; Milidiú et al., 2001) que utiliza agentes reativos e evolutivos para identificar interdependência de produtos para promoção em um mercado varejista; (ii) Uma ferramenta (Bevilacqua et al., 2001) baseada em agentes para criar grupos de consumo, e (iii) Uma ferramenta (Ribeiro, 2001) de negociação entre agentes. Através desse desenvolvimento, os pontos em comuns foram identificados e o domínio especificado.

Conseqüentemente, o domínio desse *framework* orientado a objetos engloba todas as aplicações baseadas em agentes assíncronos e distribuídos. A seção 3.1 apresenta o estudo de caso *Bundles.com* (Sardinha, 2001; Milidiú et al., 2001). A seção 3.2 apresenta o *framework* orientado a objetos ASYNC, a seção 3.3 descreve o processo de instanciação do *framework*, e a seção 3.3 apresenta uma ferramenta para executar os agentes criados com o ASYNC. Uma aplicação chamada Babilônia (Ribeiro, 2001) é apresentada de maneira resumida na seção 3.4. Esta aplicação foi implementada pela aluna de mestrado Paula Clark e documentada em sua dissertação. Além de apresentar um exemplo de re-utilização de código do ASYNC por uma outra pessoa, essa aplicação foi muito importante no processo de generalização e *design* final do *framework.*

3.1. Estudo de Caso - A Aplicação Bundles.com

A primeira aplicação que utilizou o ASYNC permite a criar promoções utilizando o conceito de agregação de produtos. A aplicação se chama *Bundles.com* (Sardinha, 2001; Milidiu et al, 2001). A tarefa de criar pacotes não é

trivial, pois muitas vezes o que parece ter apelo de marketing pode não gerar a receita esperada.

Esse projeto de software utiliza a técnica orientada a objetivos para a fase de requisitos. Esta fase utiliza o processo recursivo de decomposição de um objetivo do problema principal em vários objetivos de subproblemas mais simples.

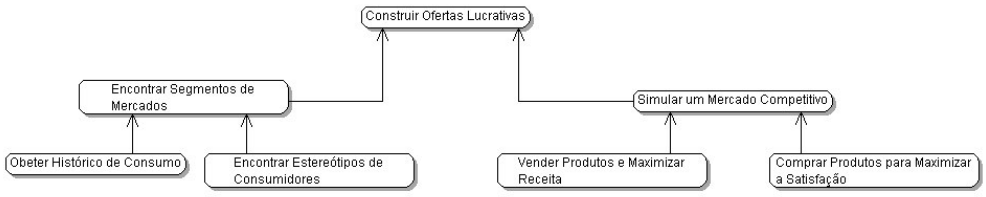

Figura 15: A decomposição dos objetivos.

O objetivo maior do sistema é encontrar pacotes de produtos que maximizem a margem de lucro. Porém, dois objetivos foram identificados para encontrar essas ofertas: (i) Encontrar segmentos de mercado através de uma base de dados e algum algoritmo de *clustering*, e (ii) Simular um mercado competitivo para encontra as ofertas mais rentáveis.

A segunda fase do processo de modelagem é criar os tipos de agentes que sejam capazes de conduzir os objetivos da figura 15. Essa criação começa com processo de relacionamento de todos os objetivos de nível mais baixo e tipos de agentes. A tabela 3 apresenta um mapeamento entre esses objetivos e tipos de agentes. A figura 16 apresenta os vários tipos de agentes e seus relacionamentos. Esta figura permite uma visão estática do projeto do nosso sistema multi-agente.

Objetivo	**Agente**
Obter Histórico de Consumo	Analista de Mercado
Encontrar Estereótipos de Consumidores	Analista de Mercado
Vender Produtos e Maximizar Receita	Vendedor
Comprar Produtos para Maximizar a Satisfação	Comprador

Tabela 3: A mapeamento entre subproblemas e tipos de agentes.

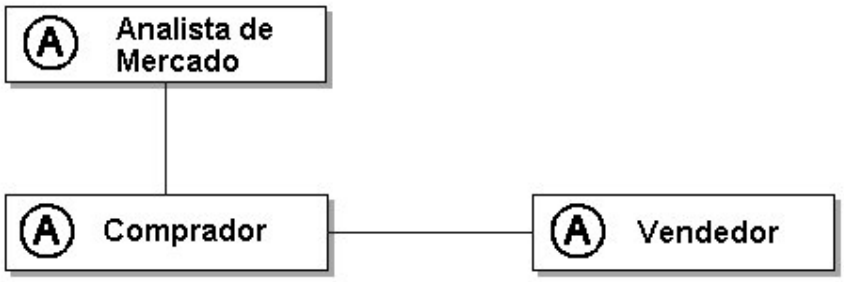

Figura 16: apresenta a arquitetura do Bundles.com.

O agente Analista de Mercado identifica os segmentos de mercado com padrões de consumo semelhantes utilizando um algoritmo de *clustering* chamado *Isodata* (Duda et al., 1973; Lebart et al., 1997). O resultado desse algoritmo são centróides que representam estereótipos de consumidores. Conseqüentemente, todos os consumidores de um mesmo segmento devem diferir muito pouco desse estereótipo. O agente executa esse algoritmo um número *N* de vezes, e seleciona apenas os estereótipos que aparecem em todas as rodadas. A figura 17 apresenta o diagrama de cenário do agente Analista de Mercado. Este diagrama possui informações sobre o contexto de como o agente deverá atingir seus objetivos.

Obter Histórico e Encontrar Estereótipos	
Agente Principal	Analista de Mercado
Pré-Condições	Evento do ambiente externo
Plano Principal das Ações	Enquanto VERDADE Ler Informações sobre Clientes no CKB Encontrar Estereótipos de Clientes Salvar Resultado no CKB Enviar mensagem para o Comprador Fim Enquanto
Interação	Comprador
Variante do Plano	

Figura 17: Diagrama de Cenários do Analista de Mercado.

A tabela 4 ilustra um resultado encontrado após as várias rodadas do algoritmo *Isodata*. As colunas dessa tabela representam os nove estereótipos encontrados a partir de uma base de dados com vários consumidores e suas transações de compras. As linhas representam a quantidade de produtos que esses consumidores estereótipos gostam de comprar para um dado período de tempo.

Através dos centróides é possível descobrir também algumas informações relevantes a respeito dos padrões de consumo dos vários consumidores dentro dos segmentos. A tabela 5 resume esse tipo de análise feito pelo agente. Os padrões de consumo são identificados através da relação dos segmentos com as classes de produtos.

	Segmentos								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Classe I	0	90	0	0	0	0	10	10	10
Classe II	0	0	0	0	0	0	0	2	0
Classe III	0	3	0	1	70	5	5	3	10
Classe IV	0	0	0	0	0	0	0	0	0
Classe V	0	0	0	0	0	0	0	7	0
Classe VI	0	0	0	0	10	0	0	13	2
Classe VII	0	0	0	0	0	0	0	0	0
Classe VIII	4	5	4	93	10	20	10	11	20
Classe IX	0	0	90	0	0	0	0	3	3
Classe X	0	0	2	2	10	70	5	3	5
Classe XI	96	2	4	5	0	0	0	7	50
Classe XII	0	0	0	0	0	5	70	36	0
Classe XIII	0	0	0	0	0	0	0	0	0
Classe XIV	0	0	0	0	0	0	0	0	0

Tabela 4: Exemplos de centróides encontrados com o algoritmo Isodata.

Segmento	Padrão de Consumo
1	Apenas produtos da classe XI.
2	Apenas produtos da classe I.
3	Apenas produtos da classe IX.
4	Apenas produtos da a classe VIII.
5	Produtos da classe III.
6	Produtos da classe X.
7	Produtos da classe XII.
8	Produtos da classe XII.
9	Produtos da classe XI.

Tabela 5: Tabela com os padrões de consumo dos nove centróides.

O agente Comprador e agente Vendedor simulam um mercado competitivo para encontrar ofertas de produtos com alta lucratividade. O resultado dessa

simulação permite descobrir mais três informações relevantes: o segmento alvo de uma oferta, a receita gerada, e o preço dos produtos.

Os Compradores recebem os resultados do Analista de Mercado para simular o comportamento de consumo. Esses agentes iniciam o processo de simulação com uma receita para comprar pacotes de bens, e utilizam a tabela 4 para selecionar os pacotes de bens que lhe interessam. A figura 18 apresenta o diagrama de cenários do Comprador.

Comprar Produtos	
Agente Principal	Comprador
Pré-Condições	Evento do ambiente externo
Plano Principal das Ações	Enquanto VERDADE Ler Tabela de Preferencias no CKB Recebe Proposta do Vendedor Decide Aceitar Proposta do Vendedor Enviar mensagem com Resultado para o Vendedor Fim Enquanto
Interação	Vendedor
Variante do Plano	

Figura 18: Diagrama de Cenários do Comprador.

Os agentes Vendedores foram modelados com características evolutivas (Winston, 1992) para descobrir pacotes que maximizem a preferência desses consumidores e a lucratividade. A construção de boas ofertas para um agente Vendedor envolve um problema de otimização combinatória. Conseqüentemente, a função de *fitness* desse agente pode ser medida pela receita que se consegue obter com a venda das ofertas menos o custo das mercadorias.

O resultado da simulação não deve encontrar apenas a melhor oferta, mas o conjunto das melhores ofertas. Conseqüentemente, o algoritmo de otimização escolhido precisa ser capaz de obter o ótimo global, e também os vários ótimos locais existentes. Este foi um dos motivos principais para a escolha de um algoritmo evolutivo para o agente Vendedor. A figura 19 exemplifica uma função de *fitness* com o seu ótimo global e seus vários ótimos locais.

Figura 19: Os máximos locais e globais da função de fitness.

A simulação desse mercado competitivo possui vários agentes vendedores que competem entre si. O objetivo de cada um é vender o maior número possível de ofertas para um número limitado de consumidores. Além disso, estes consumidores são limitados em recursos financeiros. A figura 20 apresenta o diagrama de cenários do agente Vendedor.

Vendedor Produtos	
Agente Principal	Vendedor
Pré-Condições	Evento do ambiente externo
Plano Principal das Ações	Enquanto VERDADE Reprodução Mutação Envia Proposta para Comprador sobre Oferta Recebe Resposta da Proposta e Receita Verifica se tem Receita suficiente para sobreviver Verifica se atingiu meta de Receita Fim Enquanto
Interação	Comprador
Variante do Plano	

Figura 20: Diagrama de Cenários do Comprador.

A simulação começa com agentes vendedores sem receita, porém com ofertas. Os agentes consumidores possuem dinheiro em uma conta corrente, e estão dispostos a consumir essas ofertas para maximizarem a sua satisfação. Além disso, o agente vendedor precisa acumular receita ou "energia" para sobreviver no sistema.

O agente que não consegue acumular receita após um determinado período simplesmente morre tal como uma seleção natural. O agente passa pelos seguintes estados em cada geração:

(i) Estado 1 (Reprodução) - Reproduz com outros agentes;

(ii) Estado 2 (Mutação) - Escolhe se vai criar um descendente mutante;

(iii) Estado 3 (Tenta obter Energia) - Tenta vender a oferta para os consumidores. A receita gerada pela venda representa a energia obtida em uma geração;

(iv) Estado 4 (Decide se consegue sobreviver) – O próprio agente verifica se é possível sobreviver para a próxima geração, dado um parâmetro do sistema que representa o mínimo de energia para a sobrevivência no ambiente. O agente pode não sobreviver também caso ele tenha atingido o número máximo de gerações permitidas no sistema. Esse parâmetro de entrada existe para poder estabelecer um número fixo de gerações em que o agente tenta obter a meta de energia;

(v) Estado 5 (Decide se atingiu a meta de energia) - Decide se obteve a meta de energia estabelecida. Essa meta é um parâmetro de entrada no sistema;

(vi) Estado 6 (Vida Eterna) - Quando o agente consegue atingir a meta de energia, ele apenas permanece no sistema para que outros agentes possam utilizá-lo para fins de reprodução;

(vii) Estado 7 (Morte do Agente) – O agente sai do sistema, e devolve a receita obtida aos segmentos.

3.1.1.Simulação e Monitoramento

Uma interface de monitoração *on-line* do sistema foi desenvolvida para monitorar a evolução do sistema multi-agente. Esta interface permite a configuração inicial dos parâmetros do sistema. Essa parametrização se torna disponível para os agentes através de um espaço de memória compartilhada. As informações *on-line* são recolhidas também através dessa memória compartilhada. A interface de monitoração é apresentada na figura 21.

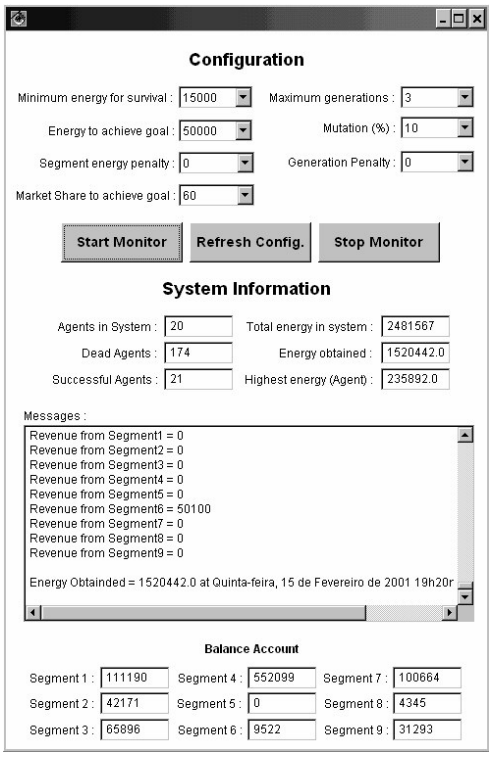

Figura 21: Interface da Monitoração On-line.

3.1.2. Resultados da Evolução do Sistema

As ofertas iniciais para os agentes vendedores são escolhidas de forma aleatória. Os agentes vendedores passam então por todos os estados descritos na figura 20 para tentar sobreviver no sistema. O gráfico da figura 22 apresenta um resultado típico da energia (receita) obtida pelos agentes vendedores nos vários instantes de tempo. As fases ilustram bem a evolução do sistema:

Fase 1 - Nessa fase há uma abundância de recursos, pois os agentes Compradores ainda não compraram nada e possuem muito dinheiro em conta corrente. Devido ao seu comportamento guloso, a heurística criada permite que os segmentos aceitem qualquer oferta interessante, mesmo não sendo a melhor.

Fase 2 - Após a fase de abundância vem um período de escassez. Na primeira fase, os Compradores compraram muitas ofertas e suas respectivas contas correntes diminuíram muito. É nesta fase que os Vendedores precisam evoluir para conseguirem vender os seus produtos.

Fase 3 - Após um período de evolução, os Vendedores conseguem novamente vender a sua cesta de produtos para os Compradores. Conseqüentemente, os Compradores esgotam quase todo o dinheiro disponível nas contas correntes. A figura 22 apresenta um gráfico da evolução do sistema, onde o eixo horizontal representa o tempo, e o eixo vertical representa o total de receita obtido por todos os Vendedores.

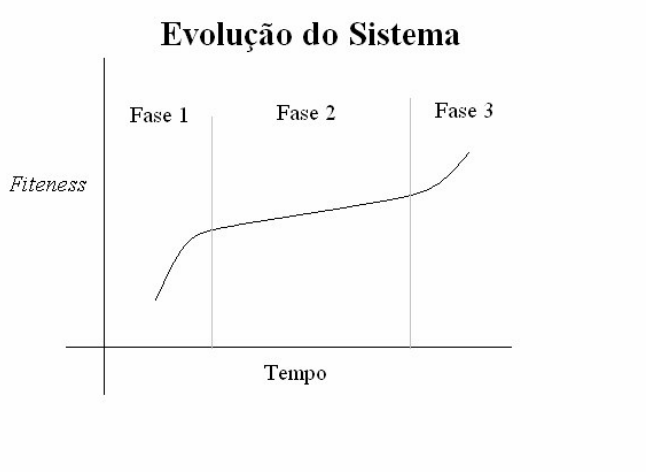

Figura 22: Energia obtida pelos Agentes por instante de tempo.

3.2. ASYNC – Um Framework Orientado a Objetos para Construção de Agentes Distribuídos e Assíncronos

O *framework* orientado a objetos ASYNC foi utilizado na construção de todas as aplicações de sistemas baseados em agentes apresentados nesta tese. O *design* desse *framework* orientado a objetos separa duas abstrações importantes em agentes: ações internas e protocolos de interação. As ações internas são tarefas executadas pelo agente que não dependem da interação com outros agentes. Os protocolos de interação definem como o agente deve se comunicar e interagir com outros agentes no sistema. Um plano do agente pode ser composto de ações internas e protocolos de interação. O *framework* orientado a objetos também fornece uma infra-estrutura de comunicação para agentes em um ambiente distribuído.

O *framework* é composto por: (i) Duas classes abstratas, *Agent* e *InteractionProtocols*; (ii) Duas classes concretas e finais, *ProcessMessageThread*

e *AgentCommunicationLayer*; e, (iii) Três interfaces, *AgentMessage, AgentBlackBoardInfo* e *AgentInterface*.

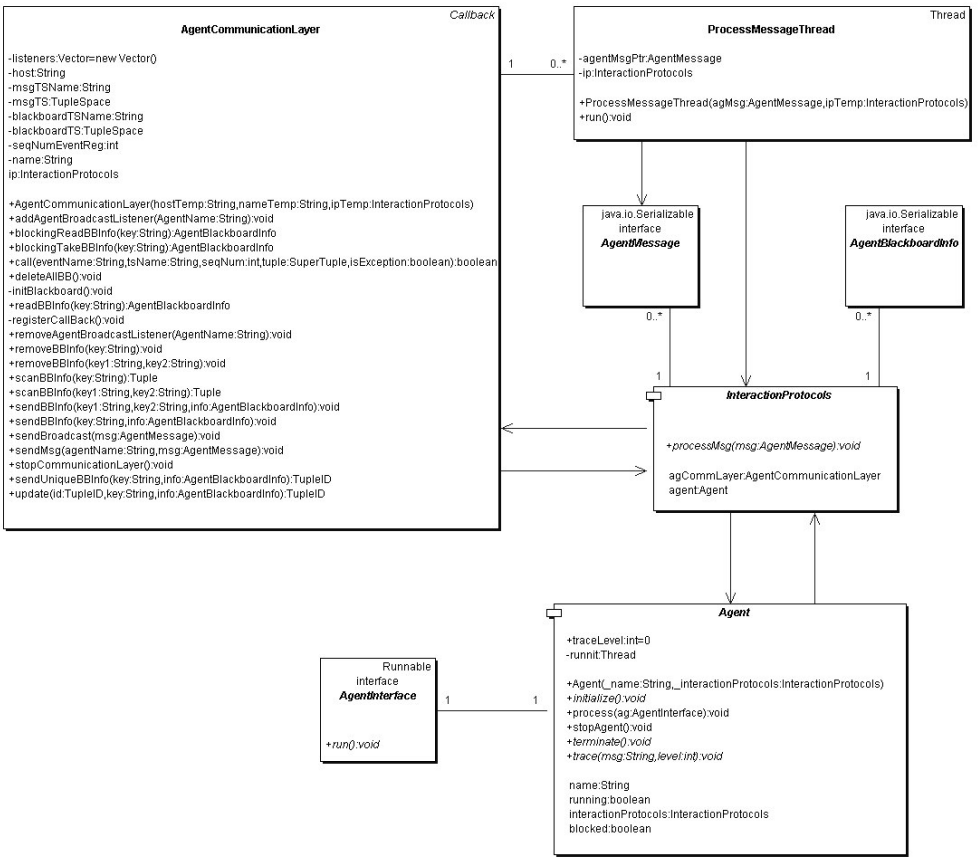

Figura 23: O diagrama de Classes do Framework Orientado a Objetos ASYNC.

A classe abstrata *Agent* fornece as funcionalidades básicas do agente: (i) *init()* - iniciar o agente e obter seus recursos do ambiente, (ii) *terminate()* – liberar os recursos do ambiente, (iii) *stop()* – parar a execução do agente, (iv) *process()* - execução das ações do agente, e (iv) *trace()* – impressão de mensagens na console. O atributo *name* desta classe guarda o nome do agente, e deve ser único no sistema por questões de implementação.

A interface *AgentInterface* é responsável por transformar a subclasse de *Agent* em uma *thread*. Essa subclasse deve implementar o método *run()*, que é responsável pela lógica da execução das ações internas do agente ou iniciar um protocolo de interação.

A classe abstrata *InteractionProtocols* define as várias maneiras como o agente deve interagir com outros agentes no sistema. Todo código relacionado com interação deve ser colocado na subclasse de *InteractionProtocols*. Essa

subclasse deve implementar o método *processMsg()*, e definir a implementação do tratamento de toda mensagem recebida por este agente.

A classe concreta e final *ProcessMessageThread* é responsável por processar as mensagens recebidas pelo agente e executar o método *processMsg()* da subclasse de *InteractionProtocols*.

A interface *AgentMessage* deve ser implementada pela classe que especifica o formato da mensagem. Por exemplo, essa classe pode especificar o formato FIPA ACL, KQML, etc. A interface *AgentBlackBoardInfo* deve ser implementada pela classe que especifica o formato das informações no espaço de tuplas do sistema.

A classe concreta e final *AgentCommunicationLayer* implementa toda a camada de comunicação do agente. Esta classe fornece uma infra-estrutura para o agente enviar mensagens síncronas ou implementar uma arquitetura de *blackboard* utilizando o espaço de tuplas distribuído. Essa classe foi implementada como uma camada sobre o software IBM TSpaces (*http://www.almaden.ibm.com /cs/TSpaces/*). O software IBM TSpaces é uma arquitetura reflexiva de espaços de tuplas em ambientes distribuídos, e fornece operações básicas no espaço para leitura, escrita, etc. Uma descrição resumida de cada método da classe *AgentCommunicationLayer* é apresentada abaixo:

(i) *addAgentBroadcastListener()* – Método para incluir um agente na lista de *broadcast*;

(ii) *blockingReadBBInfo()* – Esse método é utilizado para ler uma mensagem do espaço de tuplas. Se o agente executar esse método e a mensagem não estiver presente no espaço de tuplas, a execução do método é bloqueada até que essa mensagem esteja presente ou até um *timeout* expirar;

(iii) *blockingTakeBBInfo()* - Esse método é utilizado para ler e remover uma mensagem do espaço de tuplas. Se o agente executar esse método e a mensagem não estiver presente no espaço de tuplas, a execução do método é bloqueada até que essa mensagem esteja presente ou até um *timeout* expirar;

(iv) *deleteAllBB()* – Método utilizada para apagar todo o espaço de tuplas;

(v) *readBBInfo()* – Método utilizado para ler uma mensagem do espaço de tuplas. Se a mensagem não estiver presente, o método retorna *null*;

(vi) *removeAgentBroadcastListener()* – Método que remove o agente da lista de *broadcast*;

(vii) *removeBBInfo()* – Método para remover uma informação do espaço de tuplas. Se a mensagem não estiver presente, o método retorna *null*;

(viii) *scanBBInfo()* – Método para fazer uma pesquisa nas mensagens presentes no espaço de tuplas;

(ix) *sendBBInfo()* – Método para enviar uma mensagem para o espaço de tuplas;

(x) *sendBroadcast()* – Método para enviar uma mensagem para todos os agentes cadastrados na lista de *broadcast*;

(xi) *sendMsg()* – Método para enviar uma mensagem para um único agente;

(xii) *stopCommunicationLayer()* – Método para finalizar a camada de comunicação.

O *design* de um *framework* orientado a objetos pode ser dividido em duas partes (Fontoura et al., 1998): (i) um subsistema *kernel* ou *frozen spot* com o código comum a todas as aplicações instanciadas; e (ii) um subsistema *hot spot* que define as diferentes características das aplicações instanciadas. Os *hot spots* do ASYNC são as classes *Agent*, *InteractionProtocol*, *AgentMessage*, *AgentBlackboardInfo* e *AgentInterface*. Os *frozen spots* do ASYNC são as classes *AgentCommunicationLayer* e *ProcessMessageThread.*

3.2.1.Como Instanciar uma Aplicação Baseada em Agentes com o Framework Orientado a Objetos do ASYNC

A primeira fase do desenvolvimento de um sistema baseado em agentes é a construção do ambiente e seus recursos. O ambiente é implementado como uma classe principal do sistema para armazena todas as referencias para: (i) todos os agentes, e (ii) os recursos do sistema. A tabela 6 resume o processo de instanciação do *framework* orientado a objetos ASYNC.

Conceitos de Agentes	Classes do ASYNC instanciadas
O ambiente e os recursos	(i) Classe Principal para o ambiente; (ii) Classes para os recursos.
Agente	(i) Uma classe concreta que herda de *Agent*, e implementa *AgentInterface*; (ii) Uma classe concreta que herda de *InteractionProtocols*.
Ações internas do Agente	Métodos incluídos na classe concreta de *Agent*.
Protocolos de Interação	Método incluídos na classe concreta de *InteractionProtocols*.
Formato das Mensagens nos protocolos de interação	(i) Classe concreta que implementa *AgentMessage*, se a troca de mensagens é síncrona; (ii) Classe concreta que implementa *AgentBlackBoardInfo*, se a troca de mensagens é assíncrona.

Tabela 6: Mapeamento dos Conceitos de Agentes para as Classes da Aplicação instanciada.

Cada agente no sistema deve possuir duas classes concretas. A primeira classe a ser implementada deve herdar da classe *Agent* e implementar a interface *AgentInterface*. Essa classe concreta deve conter métodos que representam as ações internas dos agentes. Os métodos *initialize()*, *run()*, *terminate()*, e *trace()* devem ser implementados nesta classe concreta.

Esse mesmo agente do sistema deve ter outra classe concreta. Essa classe deve herdar da classe *InteractionProtocols*, e deve conter métodos para implementar os protocolos de interação. O método *processMsg()* deve ser implementado para processar todas as mensagens recebidas pelo agente. A referência para o objeto *AgentCommunicationLayer* é utilizada para que o agente possa utilizar a camada de comunicação.

3.3.Instanciando o ASYNC para a Aplicação Bundles.com

A figura 24 apresenta o diagrama de classes do agente Vendedor na aplicação *Bundles.com* instanciado com o ASYNC. As classes *SellerAgent* e *SellerAgentIP* possuem o código do agente, e esta seção apresenta os detalhes desta implementação.

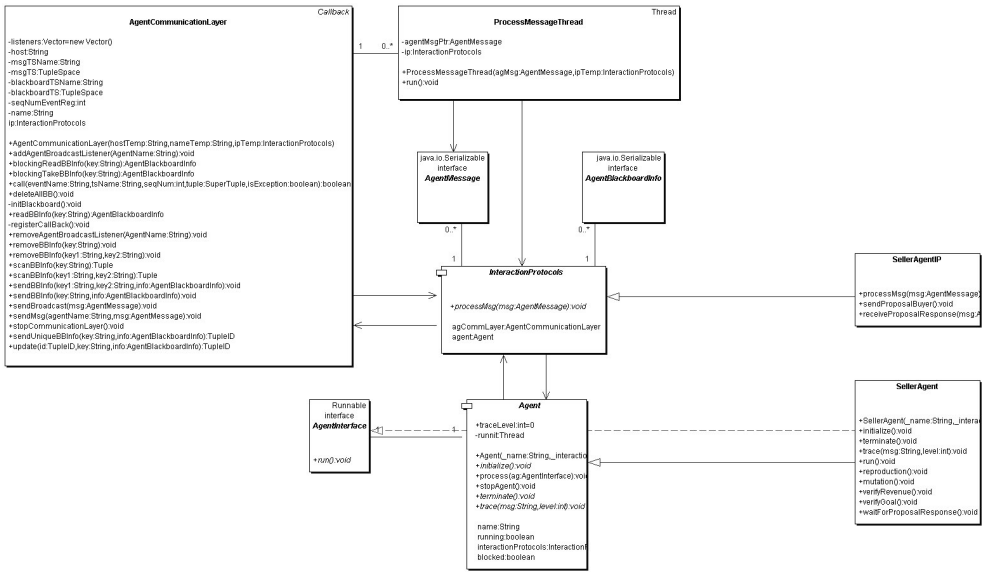

Figura 24: A instanciação do Agente Vendedor na aplicação Bundles.com.

As ações internas do Vendedor foram modeladas no diagrama de cenários do ANote da figura 25. Essas ações do agente não dependem da interação com outros agentes no sistema. As ações "Reprodução", "Mutação", "Verifica se tem Receita suficiente para sobreviver", e "Verifica se atingiu meta de Receita" não dependem da interação com outros agentes. Conseqüentemente, os métodos *reproduction()*, *mutation()*, *verifyRevenue()*, e *verifyGoal()* da classe *SellerAgent* modelam as ações descritas acima. O código Java a seguir apresenta o mapeamento das abstrações em ANote para o *framework* ASYNC.

Vendedor Produtos	
Agente Principal	Vendedor
Pré-Condições	Evento do ambiente externo
Plano Principal das Ações	Enquanto VERDADE Reprodução Mutação Envia Proposta para Comprador sobre Oferta Recebe Resposta da Proposta e Receita Verifica se tem Receita suficiente para sobreviver Verifica se atingiu meta de Receita Fim Enquanto
Interação	Comprador
Variante do Plano	

Figura 25: Diagrama de cenário do Vendedor.

```
1. public class SellerAgent extends Agent implements AgentInterface {
2.
3.    public SellerAgent(String _name, InteractionProtocols _iP) {
4.          super(_name, _iP);
5.    }
6.
7.    public void initialize() {
8.    ...
9.    }
10.
11.   public void terminate() {
12.   ...
13.   }
14.
15.   public void trace(String msg, int level) {
16.   ...
17.   }
18.
19.   public void run() {
20.         SellerAgentIP sellerIP;
21.         sellerIP = (SellerAgentIP)getInteractionProtocols();
22.         while(true) {
23.               reproduction();
24.               mutation();
25.               sellerIP.sendProposalBuyer();
26.               waitForProposalResponse();
27.               verifyRevenue();
28.               verifyGoal();
29.         }
30.   }
31.
32.   public void reproduction() {
33.   ...
34.   }
35.
36.   public void mutation() {
37.   ...
38.   }
39.
40.   public void verifyRevenue() {
```

```
41.    ...
42.    }
43.
44.    public void verifyGoal() {
45.    ...
46.    }
47.
48.    public void waitForProposalResponse() {
49.    ...
50.    }
51. }
```

Os protocolos de interação do Vendedor também são modelados pelo diagrama de cenários da figura 25. Os protocolos de interação definem como o agente deve se comunicar e interagir com outros agentes no sistema. As ações "Envia Proposta para Comprador sobre Oferta" e "Recebe Resposta da Proposta e Receita" exigem uma interação com outros agentes no sistema. Conseqüentemente, essas ações devem ser colocadas nos métodos *sendProposalBuyer()* e *receiveProposalResponse(AgentMessage msg)*. O método *sendProposalBuyer()* utiliza o método *sendMsg()* da camada de comunicação para enviar as mensagens para os agentes Compradores.

```
52.   public class SellerAgentIP extends InteractionProtocols {
53.
54.         public void processMsg(AgentMessage msg) {
55.               receiveProposalResponse((FipaACLMessage)msg);
56.         }
57.
58.         public void sendProposalBuyer() {
59.               FipaACLMessage msg = new FipaACLMessage();
60.               getAgCommLayer().sendMsg("BuyerAgent",msg);
61.
62.         }
63.
64.         public void receiveProposalResponse(FipaACLMessage msg) {
65.         ...
66.         }
67.
68.   }
```

3.4. ASYNC-D – A Ferramenta de Distribuição dos Agentes em CPUs Distintas de um Rede

A ferramenta ASYNC-D auxilia o processo de execução dos agentes em um ambiente distribuído. A ferramenta possui dois agentes de software reativos chamados Diretor de Distribuição e Coordenador de Distribuição. Esses agentes são responsáveis por coordenar a execução dos outros agentes no sistema. A figura 26 apresenta os agentes da ferramenta ASYNC-D.

O agente Diretor da Distribuição é responsável por selecionar quais agentes devem ser executados em cada CPU. Após essa decisão, o Diretor envia uma mensagem para o Coordenador de Distribuição com a lista de agentes que devem ser executados. Conseqüentemente, toda CPU deve possuir um agente Coordenador de Distribuição rodando, e eles são responsáveis por executar e "matar" os agentes de sua responsabilidade. O Diretor pode enviar uma mensagem para os Coordenadores requisitando a "morte" de algum agente também. Essa ferramenta funciona apenas com agentes de software instanciados pelo ASYNC.

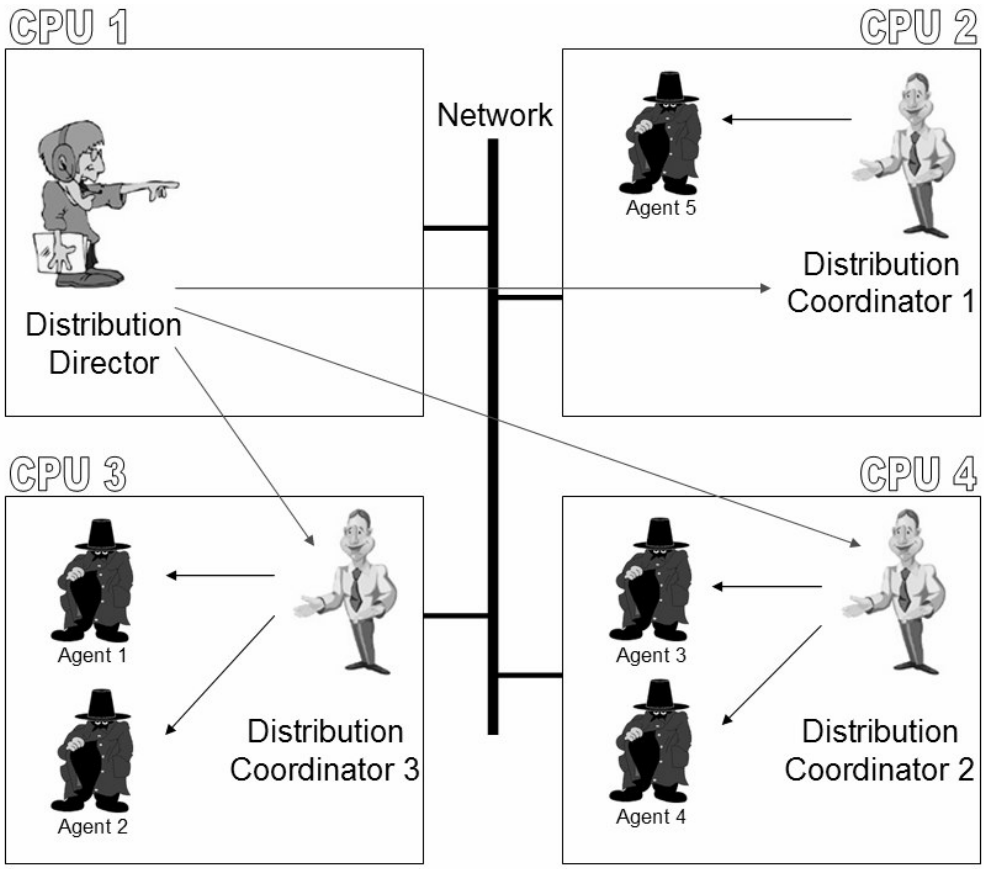

Figura 26: Os agentes do ASYNC-D.

A console de configuração da figura 27 é utilizada para definir a decisão do Diretor de Distribuição. A console permite selecionar os agentes que devem ser executados por cada Coordenador. Essa configuração é utilizada pelo Diretor para

enviar as mensagens para o Coordenador de Distribuição com a lista de agentes que devem ser executados.

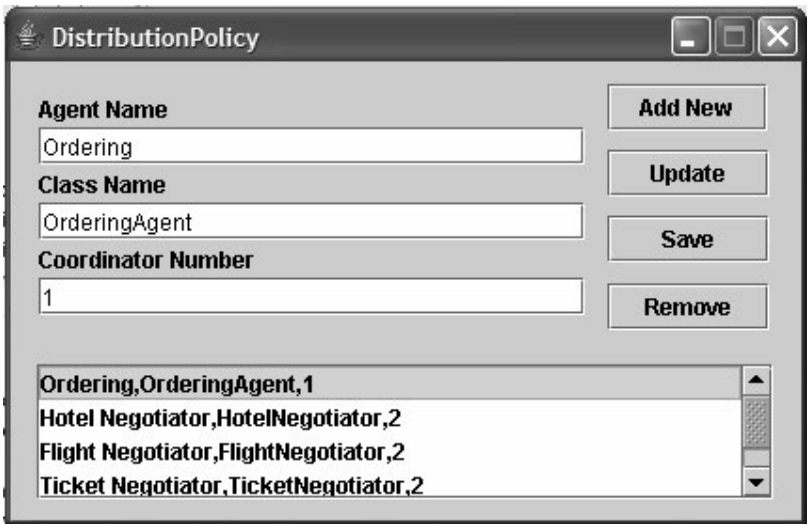

Figura 27: A Console de Configuração do ASYNC-D.

3.5. Aplicação Babilônia

Esta seção apresenta a aplicação desenvolvida pela aluna de mestrado Paula Clark em sua dissertação (Ribeiro, 2001). A modelagem da aplicação foi feita com Gaia (Wooldridge et al., 2000; Zambonelli et al, 2003), e na dissertação foi proposta um mapeamento entre os modelos de Gaia e o *framework* ASYNC.

A aplicação Babilônia é um mercado virtual para compra e venda de bens. Os agentes no sistema também implementam uma negociação automática entre compradores e vendedores. Além disso, o sistema implementa um mecanismo de certificação dos vendedores, compradores e bens para garantir a segurança das transações.

O sistema multi-agente foi modelado utilizando uma arquitetura organizacional de Gaia com os respectivos papeis de agentes: Comprador, Vendedor, e Certificador. A comprador é responsável por cadastrar as características dos bens desejados, tais como marca e modelo. Todo bem ofertado por um Vendedor que casar com esse cadastro é enviado para o Comprador. O Comprador então decide se começa ou não o processo de negociação. Na negociação alguns parâmetros são utilizados por este Comprador: preço inicial de compra, preço máximo pago pelo bem, o incremento de preço utilizado no processo de negociação.

O Vendedor também possui alguns parâmetros para o processo de negociação: preço inicial da oferta, preço mínimo que a oferta vale, e o decremento de preço utilizado no processo de negociação. O Vendedor deve indagar o agente Certificador se o processo de negociação necessita de alguma certificação.

O Certificador é responsável por certificar os Vendedores, os Compradores e os bens envolvidos no processo de negociação. Esse agente faz uma busca em vários bancos de dados e informações de proteção ao crédito fora dos sistema para garantir a segurança do processo.

O processo de negociação entre os agentes é totalmente automático, e para implementar o sistema foi utilizado o *framework* ASYNC. A figura 28 apresenta as classes do agente Comprador. A classe *AgentBuyer* especializa a classe *Agent* e implementa a interface *AgentInterface*, e a classe *AgentBuyerIP* herda da classe *InteractionProtocols*. Esse estudo de caso permitiu um grande re-uso de código, uma diminuição no tempo de desenvolvimento menor, e uma menor complexidade para implementar agentes de software.

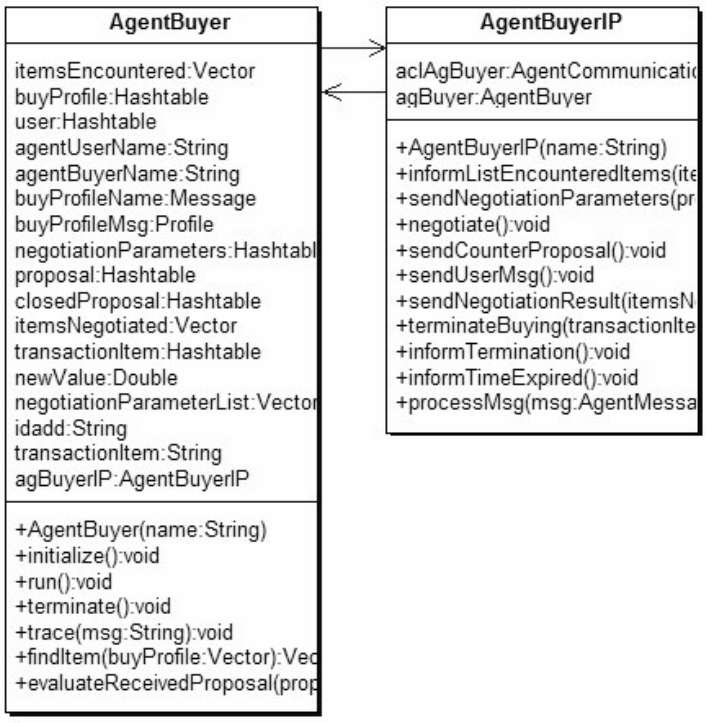

Figura 28: As classes do Agente Comprador.