UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Richard Gregory Silva Vieira

Construção de uma Interface KQML em SMALLTALK

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação

Raul Sidnei Wazlawick

Florianópolis, agosto de 2000

Construção de uma Interface KQML em SMALLTALK

Richard Gregory Silva Vieira

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação Sistemas de Conhecimento e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Raul Sidnei Wazlawick, Dr.
Orientador

Fernando A. O. Gauthier, Dr
Coordenador do Curso

Banca Examinadora

Raul Sidnei Wazlawick, Dr.

Guilherme Bittencourt, Dr.

Rogério dos Santos Bastos, Dr.

Flávio Moreira de Oliveira, Dr.

Aos meus pais, Gregório e Hélia

Para Dirlene

A todos os colegas do LSC que colaboraram com suas idéias e espírito crítico.
Ao prof. Raul S. Wazlawick, por sua orientação.

Resumo

Com a inteligência artificial distribuída e os sistemas multiagentes, surge também a necessidade de novas formas de comunicação e arquiteturas. Atualmente, já existem diversos modelos de comunicação e linguagens para a comunicação entre agentes. Dentre as várias propostas de linguagem de comunicação entre agentes, o KQML[1] tornou-se um padrão de fato[2]. Existe ainda uma outra proposta, a FIPA[3] [FIP97], a qual tem uma linguagem de comunicação entre agentes chamada ACL. Esta proposta, porém, não está tão difundida quanto o KQML.

Este trabalho busca a criação de uma biblioteca, ou melhor, de uma interface KQML para o *Squeak smalltalk*[4], que tenha como capacidade implementar o tratamento sintático e que provenha facilidades para o tratamento semântico. Propõe ainda a implementação de um tratamento de rede, que permitirá a conexão entre agentes e tratará o envio de mensagens entre os mesmos.

---

[1] KQML - Knowledge Query Manipulation Language - Linguagen de Manipulação e Consulta de Conhecimento. É uma linguagem de comunicação de agentes.

[2] Considera-se aqui o KQML como padrão de fato, pois existem diversas aplicações utilizando o KQML, entre outras o JATLite. Na internet, mais referencias podem ser encontradas em: *http://www.cs.umbc.edu/KQML/* e *http://cdr.stanford.edu/ProcessLink/papers/JATL.html.*

[3] FIPA - Foundation for Intelligent Physical Agents - Uma tentativa de especificação de mecanismo de comunicação entre agentes. Mais informações em *http://www.fipa.org*

[4] Squeak Smalltalk - O Squeak Smalltalk é uma versão gratuita de Smalltalk que tem versões para diversas plataformas de Equipamentos e Sistemas Operacionais, entre eles Windows Intel, Linux, Sparc, AIX e PDAS (Assistentes digitais pessoais, também conhecidos como Pocket Computers)

Abstract

Along with the distributed artificial intelligence and the multi-agent systems, there also appears the need of new communication ways and architectures. Nowadays, several communication models and languages already exist for communication amongst agents. Amongst several proposals of communication language between agents, KQML[5] became in fact a pattern[6]. There is also another proposal, FIPA.[7][FIP97], which has a communication language between agents called ACL. This proposal though is not very much used.

The present work proposes the creation of a library, or better saying, of an interface KQML for the *Squeak smalltalk*[8], that has both the capacity to implement the syntactic treatment and to make the semantic treatment easier. It still proposes the implementation of a net treatment which allows the connection and the sending of messages between agents.

---

[5] KQML - Knowledge Query Manipulation Language - Linguagen of Manipulation and Consultation of Knowledge. It is a language of agents' communication.

[6] He is considered KQML here as pattern in fact, because several applications exist using KQML, among other JATLite. In the internet, more references can be found in: http://www.cs.umbc.edu/KQML / and http://cdr.stanford.edu/ProcessLink/papers/JATL.html.

[7] FIPA - Foundation for Intelligent Physical Agents - An attempt of specification of communication mechanism among agents. More information in *http://www.fipa.org*

[8] Squeak Smalltalk - Squeak Smalltalk is a free version of Smalltalk that has versions for several platforms of Equipments and operating systems, among them Windows Intel, Linux, Sparc, AIX and PDAS (personal digital Assistants, also known like Pocket Computers)

SUMÁRIO

1 Introdução

1.1 Apresentação

Com o advento das redes de computadores e a crescente necessidade de poder de processamento para problemas de inteligência artificial, surgiram a Inteligência Artificial Distribuída e os Sistemas Multiagentes.

Ao contrário da inteligência artificial tradicional, a inteligência artificial distribuída baseia-se na inteligência do grupo ou sociedade. Neste caso, são as interações entre os indivíduos que geram os resultados.

Pode-se comparar um sistema multiagente com um formigueiro. Os agentes, como as formigas, têm um pequeno poder de processamento ou trabalho; porém, quando organizados em sociedade, têm um poder de trabalho muito superior, ou seja, uma maior capacidade de resolução de problemas. *Oliveira* [OLI96] diz que "...se os comportamentos individuais forem devidamente organizados/coordenados, o conjunto exibirá uma inteligência maior que a soma das inteligências individuais" Eventualmente, este poder de trabalho pode ser inclusive maior que a soma do poder de trabalho de todos os indivíduos[OLI96].

Neste momento, torna-se explícita a necessidade de uma linguagem de comunicação entre os agentes. Finin[FIN95] faz algumas considerações sobre a

linguagem de comunicação entre agentes. Nesse artigo, características como forma, conteúdo, semântica, implementação, rede, ambiente e confiabilidade são exploradas.

Surgindo de um esforço do grupo KSE (Knowledge-Sharing Effort), o KQML (Knowledge Query Manipulation Language) constitui uma linguagem padrão para a comunicação entre agentes. O KQML é baseado nos *Speech Acts*[10][SEA69] e tem uma sintaxe parecida com a do *Common Lisp*[FIN95]. O KQML será o objeto de estudo deste trabalho.

1.2 Definição do Tema, do Tipo de Pesquisa e Objetivos

Este trabalho envolve a criação e implementação de uma interface KQML para a linguagem *Smalltalk*[11].

O trabalho resume-se à pesquisa bibliográfica e implementação da interface em *Smalltalk*. A implementação será feita usando o *Squeak Smalltalk*, que é gratuito e tem ampla difusão. A escolha da construção desta interface na linguagem Smalltalk, dar-se por esta ser uma linguagem puramente orientada a objetos, com um grande poder de representação e que implementa facilidades na construção de agentes pelo fato do modelo orientado a objetos e o modelo de agentes possuírem algumas semelhanças. Outro motivo que norteou a escolha desta implementação é o fato do LSC (Laboratório

---

[10] Speech Acts – (em português Atos de Fala) é um modelo formal de comunicação humana desenvolvida por filósofos e lingüistas.

de Sistemas de Conhecimento) utilizar amplamente esta linguagem, o que facilitará a criação futura de agentes em rede.

Sua importância reside na inexistência de uma interface KQML para *Smalltalk*, o que dificulta a construção de ambientes para agentes distribuídos (multiagentes). Um fator importante da implementação sugerida é a possibilidade desta implementação KQML ser utilizada individualmente, sem a necessidade de se carregar vários outros componentes juntos, como é o caso do JATLite[12] [JAT99].

Os objetivos secundários são a criação de uma interface de rede para facilitar a comunicação entre agentes e a construção de um conjunto de agentes para a criação de uma bolsa artificial de ações, que servirá aqui como exemplo de implementação.

1.3 Estrutura do Trabalho

O capítulo 2 apresenta uma introdução aos conceitos de agentes, multiagentes, linguagens de comunicação entre agentes e o KQML.

O capítulo 3 é um resumo da especificação do KQML [FIN93].

---

[11] Smalltalk - Linguagem de programação orientada a objetos, não tipada, com grande poder de representação. Como peculiaridade, no Smalltalk todos os elementos do ambiente são objetos, sendo assim uma linguagem orientada a objetos pura.

[12] JATLite - Popular ferramenta para construção de sistemas multiagentes que implementa KQML. Esta ferramenta foi desenvolvida pela Universidade de STANDFORD.

O capítulo 4 define a implementação do KQML em *Smalltalk* usando a notação OMT[RUM91][13] e descreve suas funcionalidades. Também é apresentada uma descrição detalhada dos métodos, classes e objetos envolvidos.

O capítulo 5 realiza uma discussão sobre algumas topologias possíveis.

No capítulo 6, um exemplo simples de aplicação da biblioteca de classes definida é apresentado.

No capítulo 7, são discutidas as vantagens e problemas da implementação.

No capítulo 8, a discussão refere-se ao uso de sistemas multiagentes em simulações.

No capítulo 9, é apresentada uma breve comparação entre a interface aqui proposta e a utilizada no JATLite.

---

[13] OMT (Object Modeling Technique) – É uma metodologia de modelagem de sistemas, baseada na tecnologia de orientação por objetos.

2 Fundamentação Teórica

2.1 Sistemas Multiagentes

É quesito necessário para a compreensão do que são Sistemas Multiagentes a clara compreensão do que é um Agente de Software ou simplesmente Agente.

Para Jennings e Wooldrige [JEN96], agentes são definidos com base em suas características. Um agente, por esta definição, seria um programa com as seguintes características: autonomia, habilidade social, capacidade de percepção do ambiente, proatividade. Para outros autores, o grau de importância das características, ou até mesmo a sua definição, mudam. Alguns autores colocam a autonomia como característica básica de um agente; outros, a capacidade inteligente. Uma verificação de várias definições sobre agente pode ser encontrada em Franklin e Graesser [FRA96]. Os agentes podem ser definidos ainda pela sua intencionalidade. Sobre intencionalidade, Rivero[RIV99] faz uma boa resenha:

"Um sistema intencional pode ser de primeira ou de segunda ordem. Um sistema intencional de primeira ordem tem crenças e desejos, mas não tem crenças e desejos sobre crenças e desejos. Já os sistemas intencionais de segunda ordem têm crenças e desejos sobre suas crenças e desejos e sobre as de outros agentes."[RIV99]

Os sistemas Multiagentes partem da existência de vários agentes sob uma organização ou arquitetura. Estruturas clássicas como as de quadro-negro[14] e troca de mensagens são exemplos de arquiteturas possíveis.

Na arquitetura de quadro-negro, todos os agentes se comunicam por uma única estrutura de dados central. Não existe, portanto, comunicação direta entre os agentes. Toda a comunicação passa pelo quadro-negro.

Na arquitetura de troca de mensagens, os agentes têm comunicação direta entre si. Um problema deste tipo de arquitetura é a necessidade de cada agente "conhecer" os outros agentes na sociedade, e possuir a sua identificação.

Para cada uma das arquiteturas acima descritas, a comunicação pode ainda ser síncrona ou assíncrona.

2.2 Linguagens de Comunicação entre Agentes

Uma linguagem de comunicação entre agentes está ligada às características dos agentes. Os agentes armazenam informações ou conhecimento em representações simbólicas, ou por meio de bases de conhecimento. Isto exige uma maior capacidade de expressão da linguagem de comunicação entre os agentes e independência de implementação, considerando-se uma sociedade heterogênea.

---

[14] Quadro-Negro – Na literatura encontra-se mais facilmente o termo *Black-Board*.

Finin [FIN95] considera que uma linguagem de comunicação entre agentes tem que se adequar a características como forma, conteúdo, semântica, implementação, rede, ambiente e confiabilidade. No mesmo artigo, ele define as características:

1. **Forma**: Uma boa linguagem de comunicação entre agentes deve ser declarativa, sintaticamente simples e legível. Deve ser concisa e fácil de gerar e analisar. Deve ser independente do mecanismo de transporte e facilmente traduzida para forma linear.

2. **Conteúdo**: Uma linguagem de comunicação entre agentes deve ter camadas, onde o conteúdo é independente da linguagem de comunicação. O conteúdo deve ser encapsulado pela linguagem de comunicação, ficando para a linguagem apenas o controle dos atos de comunicação.

3. **Semântica:** A semântica de uma linguagem de comunicação entre agentes deve ser simples. Os usuários desta linguagem devem conhecê-la e entendê-la. A semântica de descrição deve prover um modelo de comunicação.

4. **Implementação**: A implementação deve ser eficiente, tanto em velocidade quando em utilização de Banda. A interface deve ser de uso fácil e os detalhes da camada de rede devem ser vistos como primitivas de comunicação, escondidas do usuário. Esta linguagem também deve permitir a implementação de subconjuntos, pois pequenos agentes terão necessidade de comportá-la.

5. **Rede**: Uma linguagem de comunicação deve trabalhar bem sobre as redes de computadores atuais. Deve permitir e suportar comunicação ponto-a-ponto, *multicast*[15] e *broadcast*[16]. Ambos os modos de comunicação devem ser suportados; assim, comunicação assíncrona e síncrona serão possíveis. A linguagem de comunicação também deve ser independente de protocolo de transporte. Assim, TCP/IP, e-mail, http, ftp, etc. podem ser usados.

6. **Ambiente**: O ambiente de um agente inteligente é, em geral, distribuído, heterogêneo e muito dinâmico. Uma linguagem de comunicação entre agentes requer um grande dinamismo e capacidade de suportar toda a heterogeneidade de um ambiente multiagentes. A linguagem deve suportar o descobrimento de conhecimento em redes.

7. **Confiabilidade**: A linguagem de comunicação deve suportar uma comunicação confiável e segura. Deve prover segurança e privacidade quando da comunicação entre dois agentes. A linguagem de comunicação deve ter meios para autenticar os agentes. Deve ter também robustez para mensagens inapropriadas ou mal formadas. Finalmente, deve ter mecanismos para assinalar e identificar erros e avisos.

---

[15] Multicast – Na língua portuguesa não tem uma tradução adequada, significa difundir par vários ou transmitir a grupos.

[16] Broadcast – Na língua portuguesa a melhor tradução é: por irradiação.

2.3 Knowledge Query Manipulation Language (KQML)

A linguagem KQML é uma linguagem de comunicação com grande força de expressão, independentemente do conteúdo e da ontologia (área de conhecimento à qual um diálogo está ligado). O KQML é uma linguagem que independe da camada de transporte, suportando tanto TCP/IP como e-mail, tanto FTP quanto HTTP.

O KQML tem três camadas[FIN95]: a camada de conteúdo, a camada de mensagem e a camada de comunicação.

A camada de conteúdo pode conter qualquer tipo de informação. O KQML apenas vai realizar o transporte da informação, não efetuando qualquer outra operação sobre o seu conteúdo. Alguns agentes, como roteadores[17] ou facilitadores[18], podem até mesmo ignorar a sua existência, a não ser pela necessidade de saber onde o conteúdo termina.

A camada de comunicação implementa um conjunto de características que descrevem parâmetros para a comunicação. Parâmetros estes como *sender* (define o remetente) e *receiver* (define o destinatário), entre outros, para identificar de forma única a comunicação.

A camada de mensagem é responsável por codificar a mensagem e transmiti-la ao outro agente e forma o núcleo da linguagem KQML. A função primária da camada

---

[17] Roteadores – neste texto, agentes roteadores ou apenas roteadores, são agentes que realizam o trabalho de entrega roteamento (achar o agente de destino) das mensagens.

[18] Facilitadores – agentes facilitadores, são agentes que implementam diversos serviços auxiliares a comunicação, como manutenção de registro de nomes, encaminhamento de mensagens, catalogo de

de mensagem é identificar o protocolo a ser usado para transportar a mensagem e suprir um ato de fala ou *performative*[FIN95] ao qual vai ser anexado o conteúdo.

A sintaxe do KQML é baseada em uma lista balanceada de parênteses, muito semelhante ao *Common Lisp*. Na lista, o primeiro elemento que aparece é o *performative* (que define a mensagem), seguido de pares compostos por palavras-chaves/valores.

Para ilustrar, um exemplo de comunicação entre um agente "joe" e o "stock-server".

```
(ask-one
  :sender joe
  :content (PRICE IBM ?price)
  :receiver stock-server
  :reply-with ibm-stock
  :language LPROLOG
  :ontology NYSE-TICKS)
```

A mensagem acima pergunta ao *stock-server* o preço da ação IBM; para isso, usa o comando *PRICE IBM ?price*, que está escrito em *LPROLOG*.

Sender define quem está mandando a mensagem.

Content é o conteúdo da mensagem propriamente dito. Deve estar na linguagem definida em *language*.

---

fornecedores de informação e serviços, além de serviços de mediação e tradução de informações (ou conhecimento).

Receiver é o receptor da linguagem.

Reply-with é o identificador da mensagem. Quando o receptor responder, ele usará esta identificação para a resposta.

Language é a linguagem em que o conteúdo é expresso.

Ontology é o assunto ao qual a mensagem está ligada. Podemos também definir ontologia como área de conhecimento ao qual um diálogo está ligado.

Como resposta, o *stock-server* teria:

```
(tell
  :sender stock-server
  :content (PRICE IBM 14)
  :receiver joe
  :in-reply-to ibm-stock
  :language LPROLOG
  :ontology NYSE-TICKS)
```

O KQML possui diversas classes de mensagens, das mais simples às mais complexas. A tabela 1 faz uma classificação das mensagens[FIN95]:

Categoria	*Performative*
Basic query	evaluate, ask-if, ask-about, ask-one, ask-all
Mult-reponse (query)	tream-about, stream-all, eos
Response	reply, sorry
Generic informational	tell, achieve, cancel, untell, unarchieve
Generator	standby, ready, next, rest, discard, generator
Capability-definition	advertise, subscribe, monitor
Networking	register, unregister, forward, broadcast, pipe, break, transport-address

Tabela 1: Divisão de algumas *performatives* em sete categorias básicas[FIN95]

Existem no KQML mensagens de *query* simples ou de multi-resposta e de informação (geralmente para prover a resposta); porém, estão definidas também mensagens de controle de informação de capacidade, de rede, de resposta (neste caso, mensagem de erro) e de geração.

O KQML permite ainda que usemos agentes como facilitadores. Os facilitadores são geralmente agentes que conhecem os agentes registrados em uma sociedade e suas respectivas capacidades.

Desta forma, quando um novo agente é inserido em uma sociedade, ele pode reportar-se ao facilitador para avisar de suas capacidades e questionar qual agente tem um serviço, ou ainda solicitar que este intermedie o serviço.

Uma característica importante do KQML é a existência do conceito de *knowledge bases (KB)*, que é basicamente o objeto de manipulação do KQML[FIN93].

Todas as *performatives* são baseadas no conhecimento existente na base de conhecimento dos agentes, seja para perguntar o seu conhecimento, seja para designar a rota apropriada de uma mensagem para outros agentes.

Muitas implementações de agentes não estão estruturadas como uma base de conhecimento. Podem ter implementações em base de dados simples, ou usando uma estrutura de dados especial, sendo necessária uma tradução para a representação em construções baseadas em conhecimento para que todos possam se beneficiar desta construção. Esta tradução é vista como uma *Virtual Knowledge Base (VKB)* ou base de conhecimento virtual. [FIN93]

3 Especificação e Definição da Linguagem KQML

A linguagem KQML para a camada de transporte é vista como uma cadeia de caracteres. Como foi dito anteriormente, o KQML tem uma sintaxe semelhante a uma representação *Common Lisp*.

A utilização de uma cadeia de caracteres foi preferida pela facilidade de visualização por humanos e pela simplicidade ao realizar o seu parser por programas.

3.1 Sintaxe de uma Cadeia de caracteres KQML em BNF[19]

A definição BNF de uma Cadeia de caracteres KQML é sugerida por Finin et al. [FIN93]. Podemos encontrá-la na figura 1.

---

[19] BNF – Backus Naur Form – Notação formal para descrever a sintaxe de uma linguagem.

```
<performative>::= (<word> {<whitespace> :<word> <whitespace> <expression>}*)
<expression>  ::= <word> | <quotation> | <string> |
                  (<word> {<whitespace> <expression>}*)
<word>        ::= <character><character>*
<character>   ::= <alphabetic> | <numeric> | <special>
<special>     ::= < | > | = | + | - | * | / | & | ^ | ~ | _ |
                  @ | $ | % | : | . | ! | ?
<quotation>   ::= '<expr> | `<comma-expr>
<comma-expr>  ::= <word> | <quotation> | <string> | ,<comma-expr> |
                  (<word> {<whitespace> <comma-expr>}*)
<string>      ::= "<stringchar>*" | #<digit><digit>*"<ascii>*
<stringchar>  ::= \<ascii> | <ascii>-\-<double-quote>
```

Figura 1: Definição BNF DO KQML[FIN93]

3.2 Parâmetros Reservados de *Performative*

As *performatives* têm parâmetros designados por palavras-chave. Será definido agora um conjunto de parâmetros reservados, com funções específicas e iguais para todas as *performatives*. Devemos lembrar que o KQML é extensível, isto é, podem ser anexadas novas *performatives* com novos parâmetros.

Performatives:

:sender <palavra>

:receiver <palavra>

Definem quem está mandando e quem deve receber a mensagem.

:reply-with <expressão>

:in-reply-to <expressão>

Estes parâmetros identificam a mensagem com a qual está sendo realizada a conversação. Quando perguntamos algo, usamos **:reply-with** <expressão>, onde o nome em expressão será retornado junto à resposta na cláusula :in-reply-to. Assim, o agente sabe qual pergunta está sendo respondida.

:content <expressão>

:language <word>

:ontology <word>

O parâmetro **:content** possui o conteúdo da mensagem propriamente dito. O parâmetro **:language** possui o identificador da linguagem usada. O parâmetro **:ontology** possui o assunto do diálogo.

:force <word>

Se o valor do parâmetro for a palavra *permanent*, o emissor garante que nunca vai negar a *performative*. O valor default é *tentative*.[FIN93]

3.3 Nomes de *Performatives* Reservadas

Quando da definição do KQML, foi criado um conjunto de *performatives* reservadas. Estas *performatives* não podem ser redefinidas em sua semântica.

Nome	Significando
ask-if	S[20] quer saber se o :content está na VKB de R[21]
ask-all	S quer saber todas as instâncias de :content que são verdade em R
ask-one	S quer saber uma instância de :content que é verdade em R
stream-all	versão de múltiplo-resposta de perguntar-tudo
eos	o marcador de fim-de-Stream para uma múltiplo-resposta (fluxo-todos)
tell	a oração em :content está na VKB de S
untell	a oração não está na VKB de S
deny	a negação da oração está na VKB de S
insert	S pede que R insira o :content para seu VKB
uninsert	S quer que R inverta o ato de uma inserção prévia
delete-one	S quer que R remova uma oração compatível de seu VKB
delete-all	S quer que R remova todas as orações compatíveis de seu VKB
undelete	S quer que R inverta o ato de um prévio delete
achieve	S quer que R faça algo verdadeiro de seu ambiente físico
unachieve	S quer que R inverta o ato de um prévio alcance
advertise	S quer que R saiba que S pode e processará uma mensagem como a que está em :content

[20] S nesta tabela representa *SENDER* (quem esta enviando a mensagem)
[21] R nesta tabela representa *RECEIVER* (quem esta recebendo a mensagem)

Nome	Significando
unadvertise	S quer que R saiba que S cancela um prévio anúncio e não processará mais nenhuma mensagem como a que está no :content
subscribe	S quer atualizações à resposta de R para um performative
sorry	S entende mensagem de R mas não pode prover uma resposta mais informativa
standby	S quer que R anuncie sua prontidão para prover uma resposta para a mensagem em :content
ready	S está pronto para responder previamente a uma mensagem que recebeu de R
next	S pede uma próxima resposta para R
rest	S pede todas as respostas restantes para R
discard	S não quer mais as respostas restantes para R
register	S anuncia a R sua presença e nome simbólico
unregister	S quer que R inverta o ato de um registro prévio
forward	S quer que R remeta a mensagem para o agente de :to (R poderia ser aquele agente)
broadcast	S quer que R envie uma mensagem a todos os agentes que R conhece
transport-address	S associa seu nome simbólico a um endereço de transporte novo
broker-one	S quer que R ache uma resposta para uma <performative> (algum agente diferente de R vai prover aquela resposta)
broker-all	S quer que R ache todas as respostas para uma <performative> (algum agente diferente de R vai prover aquela resposta)
recommend-one	S quer aprender um agente que pode responder uma <performative>
recommend-all	S quer aprender de todos os agentes que podem responder uma <performative>
recruit-one	S quer que R consiga que um agente responda uma <performative>
recruit-all	S quer que R consiga que todos os agentes respondam uma <performative>

Tabela 2: Sumário das Performativas reservadas. Onde: :sender S e :receiver R.

4 Definição da Estrutura de Classes

A estrutura necessária para implementar a linguagem KQML se compõe de três classes:

- **KqmlMessage** – As instâncias desta classe são estruturas contendo todas as informações de uma mensagem KQML, em um formato simples de trabalhar para um agente escrito em smalltalk. As instâncias desta classe poderão ser serializadas para uma mensagem KQML propriamente dita, ou converter uma mensagem KQML para uma instância de Kqmlmessage.
- **KqmlTreatment** – Classe que implementa o tratamento da mensagem KQML. As instâncias desta classe têm a responsabilidade de realizar uma análise sintática mais profunda, verificando erros e repassando-os para que o agente realize o tratamento adequado.
- **ConnectionProxy** - Classe que implementa a comunicação entre agentes usando o TCP/IP. As instâncias desta classe prestam serviços de conexão e envio de mensagens sobre uma rede TCP/IP.
- **Agent** – É uma classe abstrata. Temos aqui definida apenas a estrutura básica para possibilitar o tratamento de mensagens KQML. Todos os atributos e operações específicas deverão ser implementadas nas suas subclasses, onde deverá também ser implementado o seu comportamento. Nesta classe é definido somente o nome, que é a sua identificação, e uma ligação a um transporte, chamado **ConnectionProxy**.

A representação OMT[22]do diagrama de classes segue abaixo:

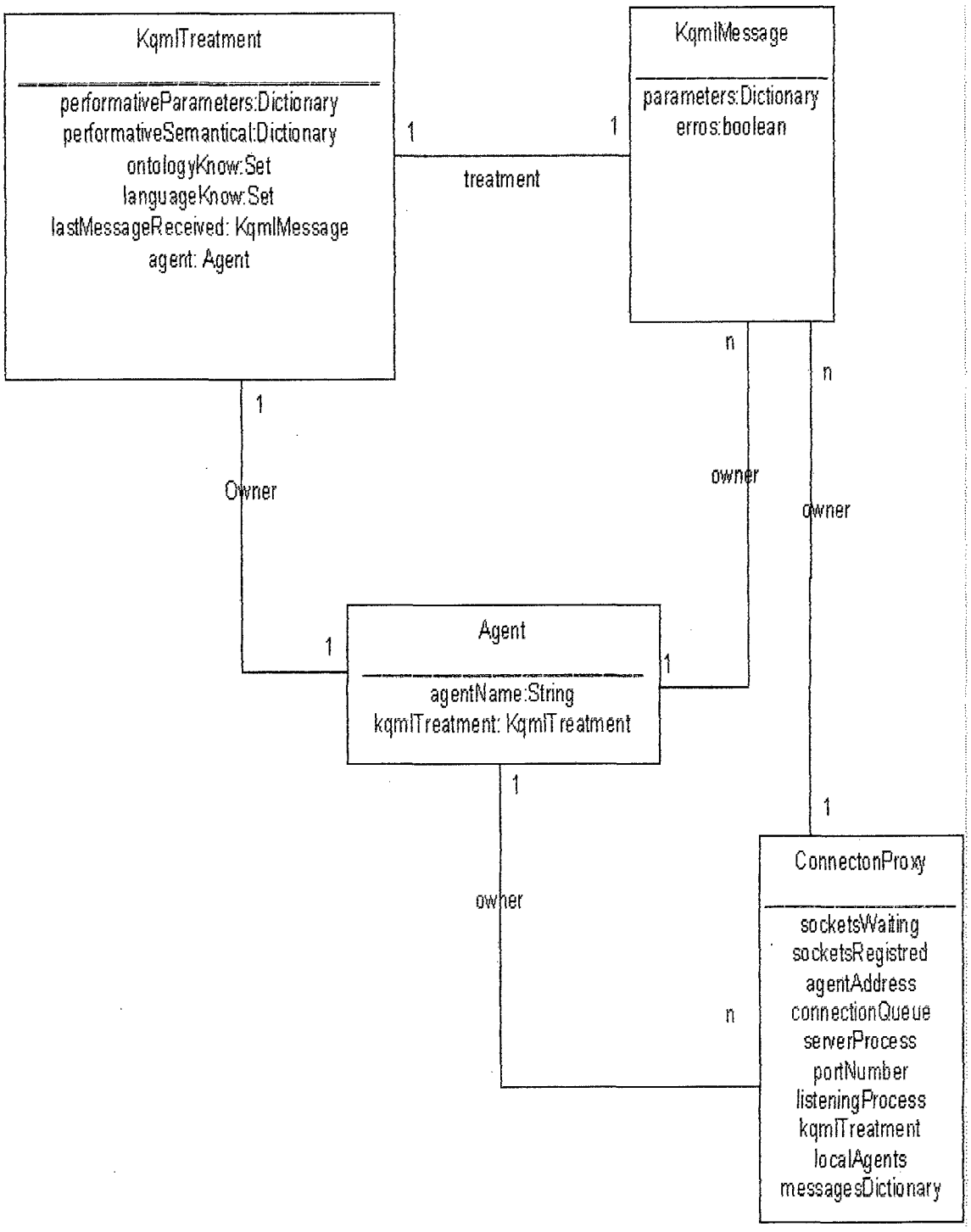


Figura 2: Representação OMT da Estrutura de Classes

[22] OMT (Object Modeling Technique) – É uma metodologia de modelagem de sistemas, baseada

4.1 Estrutura interna das classes

4.1.1 KqmlMessage

Um objeto do tipo KqmlMessage possui como estrutura um dicionário, onde são colocados os parâmetros (nome do parâmetro, valor do parâmetro), e uma cadeia de caracteres, que denota o nome da *performative*. Para os parâmetros que definem o endereçamento e a identificação da mensagem, os quais são reservados do KQML, existem métodos para a recuperação destes valores. Apenas os parâmetros definidos pelos usuários (extensões do KQML) serão repassados, em forma de um dicionário, para quem for tratar esta mensagem (no nosso caso, o objeto definido pela classe KqmlTreatment).

Como métodos desta classe, vale ressaltar:

- **messageToKqml**: *aString*

Esta mensagem faz a *desserialização* da mensagem KQML. Transforma a mensagem KQML do formato de uma Cadeia de caracteres para o formato interno do objeto KqmlMessage.

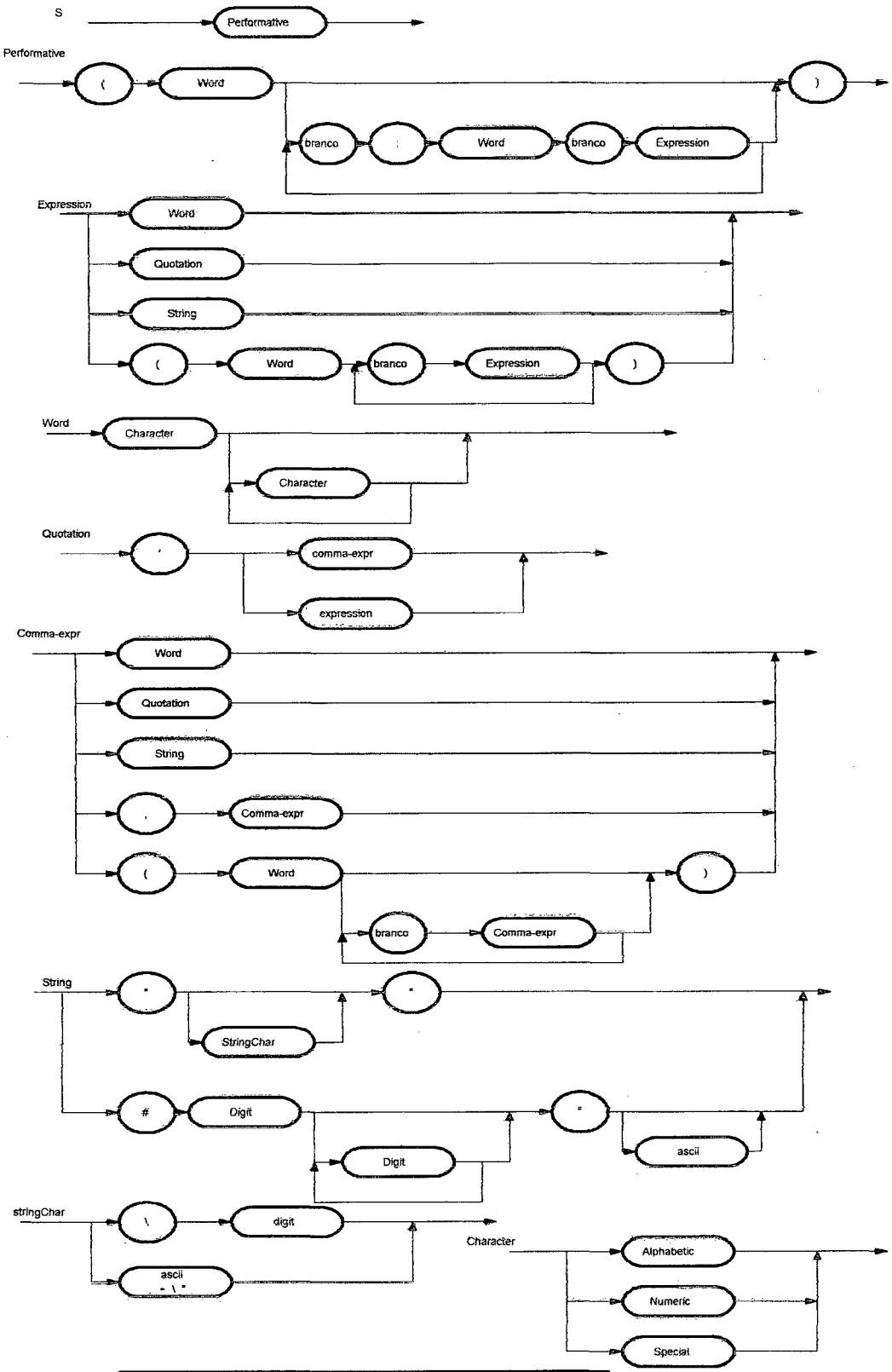
- **kqmlToString**

---

na tecnologia de orientação por objetos.

Esta mensagem faz a *serialização* do objeto *KqmlMessage* para uma cadeia de caracteres no formato de uma mensagem KQML.

Os métodos stringToMessage e messageToString realizam as conversões baseadas nas seguintes definições de gramática:



Alphabetic	Todas as Letras do Alfabeto, Maiusculas e Minusculas.
Numeric	Todos os numeros, mais os simbolos de Sinal, ponto e divisores de milhares.
Special	Os Simbolos: <, >, =, +, -, *, /, &, ^, ~, _, @, $, %, :, ., !, ?
Digit	Os Numeros: 0,1,2,3,4,5,6,7,8,9,0
Ascii	Todos os caracteres do codigo ASCII

Figura 3: Representação gráfica da gramática para o KQML

- ***performative***

  Retorna a *performative* da mensagem.

- ***performative***: *aPerformativeName*

  Define uma nova *performative* para mensagem.

- **replyWith**

  Retorna a identificação de mensagem da cláusula *reply-with.*

- **replyWith**: *aMessageId*

  Define a identificação de mensagem da cláusula *reply-with.*

- **inReplyTo**

  Retorna o valor da cláusula *in-reply-to.*

- **inReplyTo**: *aMessageId*

  Define o valor da cláusula in-reply-to.

- **content**

  Retorna o valor da cláusula *content.*

- **content**: *aContent*

  Define o valor da cláusula *content.*

- **language**

  Retorna o valor da cláusula *language.*

- **language**: *aLanguage*

Define o valor da cláusula *language*.

- **ontology**

  Retorna o valor da cláusula *ontology*.

- **ontology**: *anOntology*

  Define o valor da cláusula *ontology*.

- **force**

  Retorna o valor da cláusula *force*.

- **force**: *aCondition*

  Define o valor da cláusula *force*.

- **otherParametersName**: *aName*

  Responde com o valor associado ao parâmetro com o nome definido por *aName*.

- **otherParametersName**: *aName* **content**: *aContent*

  Especifica o parâmetro definido por *aName* com o conteúdo em *aContent*.

4.1.2 KqmlTreatment

Um objeto do tipo *KqmlTreatment* tem como função realizar um tratamento sintático e semântico de um objeto do tipo *KqmlMessage* a ele repassado.

Para que o objeto deste tipo possa realizar suas tarefas, ele deve dispor de um conjunto relativamente grande de informações, tais como: as mensagens recebidas, as

performatives conhecidas, os parâmetros aceitos para cada *performative* conhecida, a semântica de cada *performative*, as ontologias conhecidas, as linguagens conhecidas, os agentes conhecidos e as mensagens enviadas.

Teremos então as seguintes estruturas:

Dicionários:

- **performativeParameters**: Quais parâmetros são aceitos para cada *performative*.
- **performativeSemantical**: Quais atitudes o objeto deve tomar. As atitudes possíveis são: responder com *error*, responder com *sorry* ou repassar o controle para o agente dono deste objeto.
- **messageDictionary**: Quais mensagens foram enviadas. A chave neste caso é a identificação *reply-with*.
- **reponse**: Quais mensagens foram recebidas. A chave neste caso é *in-reply-to*.

Conjuntos:

- **ontologyKnow**: Ontologias conhecidas. Todas as ontologias que o agente pode tratar.
- **LanguageKnow**: Linguagens conhecidas. Todas as linguagens que o agente pode reconhecer.
- **AgentsKnow**: Todos os agentes que são conhecidos pelo agente.

Outras estruturas:

- **lastMessageReceived**: Última mensagem recebida. É a mensagem que está sendo tratada pelo objeto.
- **agent**: É o agente dono deste objeto.

Como métodos desta classe, vale ressaltar:

- **addPerformative**: *aPerformative*

  Adiciona uma *performative* na definição do agente.

- **addPerformative**: *aPerformative* **parameters**: *parameters*

  Adiciona um parâmetro na definição da *performative* do agente.

- **removePerformative**: *aPerformative*

  Remove uma *performative* da definição do agente.

- **removePerformative**: *aPerformative* parameters: parameters

  Remove um parâmetro de uma *performative* da definição do agente.

- **checkValidPerformative**: *aPerformative*

  Verifica se uma *performative* é definida pelo agente.

- **checkValidPerformative**: *aPerformative* **parameters**: *parameters*

  Verifica se um parâmetro de uma *performative* é válido na definição do agente.

- **listParametersPerformative**: *aPerformative*

  Retorna uma lista contendo os parâmetros válidos para uma *performative*

- **listPerformatives**

Retorna uma lista contendo as *performatives* válidas para um agente.

- **addSemantical**: *aPerformative* **semantical**: *anObject*

Adiciona um comportamento para uma determinada *performative*. Inicialmente, os símbolos válidos serão: *#sorry*, *#error* e *#agent*, onde *#sorry* avisa ao objeto para responder com a *performative* sorry, *#error* com a *performative* error, e *#agent* avisa que o tratamento deverá ser feito pelo agente.

- **removeSemantical**: *aPerformative*

Remove a semântica de uma *performative*. Por default e *#agent*.

- **executeSemantical**: *aPerformative*

Executa a *performative*. Caso o comportamento seja *#error* ou *#sorry*, deve ser enviada imediatamente a mensagem apropriada de resposta. Caso seja *#agent*, o tratamento é deixado a cargo do agente.

- **addOntology**: *aString*

Adiciona uma ontologia na lista de ontologia.

- **removeOntology**: *aString*

Remove uma ontologia da lista de ontologia.

- **checkOntology**: *aString*

Verifica se uma ontologia é aceita pelo agente.

- **addLanguage**: *aString*

Adiciona uma linguagem na lista de linguagens.

- **removeLanguage**: *aString*

  Remove uma linguagem da lista de linguagens.

- **checkLanguage**: *aString*

  Verifica se uma linguagem é compreendida pelo agente.

- **addAgents**: *aString*

  Adiciona um agente na lista de agentes conhecidos.

- **removeAgents**: *aString*

  Remove um agente da lista de agentes conhecidos.

- **checkAgents**: *aString*

  Verifica se um agente está na lista de agentes conhecidos.

- **listAgents**

  Lista todos os agentes conhecidos.

- **prepareMessage**: *aKqmlMessage* **replyWith**: *aString*

Faz a análise de uma mensagem do agente. Em caso de sucesso, retorna um objeto *KqmlMessage* devidamente montado; em caso de falha, retorna *nil*.

- **evaluateMessage**: *aKqmlMessage*

  Avalia uma KqmlMessage (mensagem KQML), executando-a.

- **new**: *anAgent*

Cria um *KqmlTreatment*, passando como parâmetro o objeto-agente dono da mesma.

4.1.3 ConnectionProxy

Um objeto do tipo *ConnectionProxy* tem como função realizar conexões entre dois ou mais agentes. As conexões são feitas via protocolo *TCP/IP*.

Um objeto *ConnectionProxy* é contemplado com uma série de serviços para realizar a conexão e o transporte das mensagens KQML. Estes objetos também são responsáveis pela distribuição das mensagens e sua entrega ao agente (colocação na fila de mensagens para o agente).

Um objeto *ConnectionProxy* pode atuar como cliente/servidor (de comunicação) ou somente como cliente.

Quando um objeto *ConnectionProxy* é colocado como servidor, ele passa a ouvir uma porta específica, aguardando uma conexão. Quando ocorre uma conexão, é criada uma nova instância em uma porta disponível; esta instância é mantida até o fim da conexão.

Um agente pode ter quantas instâncias de *ConnectionProxy* desejar, podendo assim ouvir mais de uma porta. Porém, o mais natural é que vários agentes, em uma mesma imagem, utilizem uma mesma instância do *ConnectionProxy*. Desta forma, a comunicação entre os agentes com a mesma *ConnectionProxy* será feita utilizando mensagens *smalltalk*, sem a necessidade de uma conexão *TCP/IP*.

A forma de utilização do *ConnectionProxy*, assim como sua topologia, depende principalmente das características e funções do agente. Carga e quantidade de conexões também podem interferir na sua topologia.

Uma característica desejável do *ConnectionProxy*, que foi negligenciada nesta implementação, é a busca automática de outras *ConnectionProxy* ativas e a recuperação e resolução dos nomes nestas *ConnectionProxy*.

Uma *ConnectionProxy* tem capacidade de "resolver nomes" dos agentes registrados localmente e externos a ela.

Teremos então as seguintes estruturas:

Dicionários:

- **socketsWaiting**: Quais parâmetros são aceitos para cada *performative*.
- **socketsRegistred:** Quais conexões são agentes conectados.
- **agentAddress:** Dicionario nome do agente -> endereço (Socket)
- **localAgents:** Agentes que estão sob a imagem deste agente.
- **messagesDictionary:** Dicionário de mensgens.

Outras estruturas utilizadas são:

- **connectionQueue**

Fila de conexões em espera. Em caso de incapacidade do *connectionProxy* de atender imediatamente todas as conexões solicitadas, estas ficam armazenadas nesta fila.

- **ServerProcess**

Processo servidor. Esta estrutura armazena o processo que cuida do recebimento de uma solicitação de conexão. Deve ser armazenado para que haja a possibilidade de controles como encerramento do processo e repriorização, além de permitir a comunicação com o processo.

- **ListiningProcess**

É uma lista de todos os processos de comunicação entre as *connectionProxy*.

Como métodos dessa classe, vale ressaltar:

- **localAgents**: *name*

Retorna, se existir, o agente localizado na mesma imagem.

- **registerLocalAgents**: *name* **agent**: *agent*

Registra um agente com nome "name" localmente. Se "name" já estiver sendo utilizado por outro agente, será retornado *false*.

- **register**: *agentName* **socket**: *aSocket* **firstId**: *firstId*

Registra um agente externo.

- **registredAgents**

Retorna uma lista com o nome de todos os agentes registrados.

- **listening**

Este método fica *ouvindo*[23] todos os *sockets* conectados. É responsável pelo recebimento das mensagens, sua separação e registro.

- **loopOnPort**

Fica *escutando*[24] uma porta previamente definida. Quando alguém tenta conexão, este dispara uma nova *socket* para atender esta conexão.

- **portNumber**: *portNumber*

Define qual porta será escutada. Caso não seja declarada nenhuma porta, ou *nil*, a *connectionProxy* será apenas cliente.

- **startListeningForConnection** e **stopListeningAllConnections**

Inicia e pára o servidor de conexões.

- **initialize**

Este método configura todos os serviços do *connectionProxy*. Caso exista uma porta definida, o *connectionProxy* começará imediatamente a ouvir a porta.

- **connect**: *agentName* **address**: *address* **port**: *port*

Abre uma conexão para um agente com nome "agentName", localizado em uma *connectionProxy* com endereço definido por "address" e porta definida por "port".

Caso não consiga realizar a conexão, retorna *nil*. É importante ressaltar que apenas é aberta uma conexão, mas sem registrar o agente.

- **getMessageQueue**: *agentName*

---

[23] O termo ouvindo aqui se refere a ficar verificando se existe alguma informação em uma determinada "porta" ou "conector" de rede.

Retorna a fila de mensagens para um determinado agente; se esta for vazia, retorna *nil*.

- **haveMessage**: *agentName*

  Retorna *true* se existir alguma mensagem para o agente.

- **sendMessage**: *kqmlMessage*

  Envia a mensagem *KQML* para o agente constante na cláusula *receiver*.

---

[24] O mesmo que ouvindo.

5 Topologias Possíveis

É possível obter várias topologias de agentes e de rede utilizando as classes citadas previamente neste trabalho. Para cada tipo de agente ou sociedade de agentes, pode-se ter topologias que serão bem mais adequadas para a resolução do problema.

5.1 Centralizado

Neste tipo de topologia, apenas um agente ou uma imagem[25] trabalha como servidor, enquanto as outras imagens se conectam à imagem centralizadora.

Sociedades típicas para este tipo de topologia são bolsas de valores ou leilões. Neste caso, a responsabilidade é de aceitar e coordenar as conexões.

A figura 4 mostra esta topologia.

---

[25] O termo "imagem" refere-se ao conceito de "maquina virtual", que é um ambiente independente, ou a simulação de uma maquina dentro de outra, a qual pode ter uma ou mais instancias. Cada instancia de uma imagem tem o seu próprio ambiente individual, desconhecendo inclusive a existência de outros.

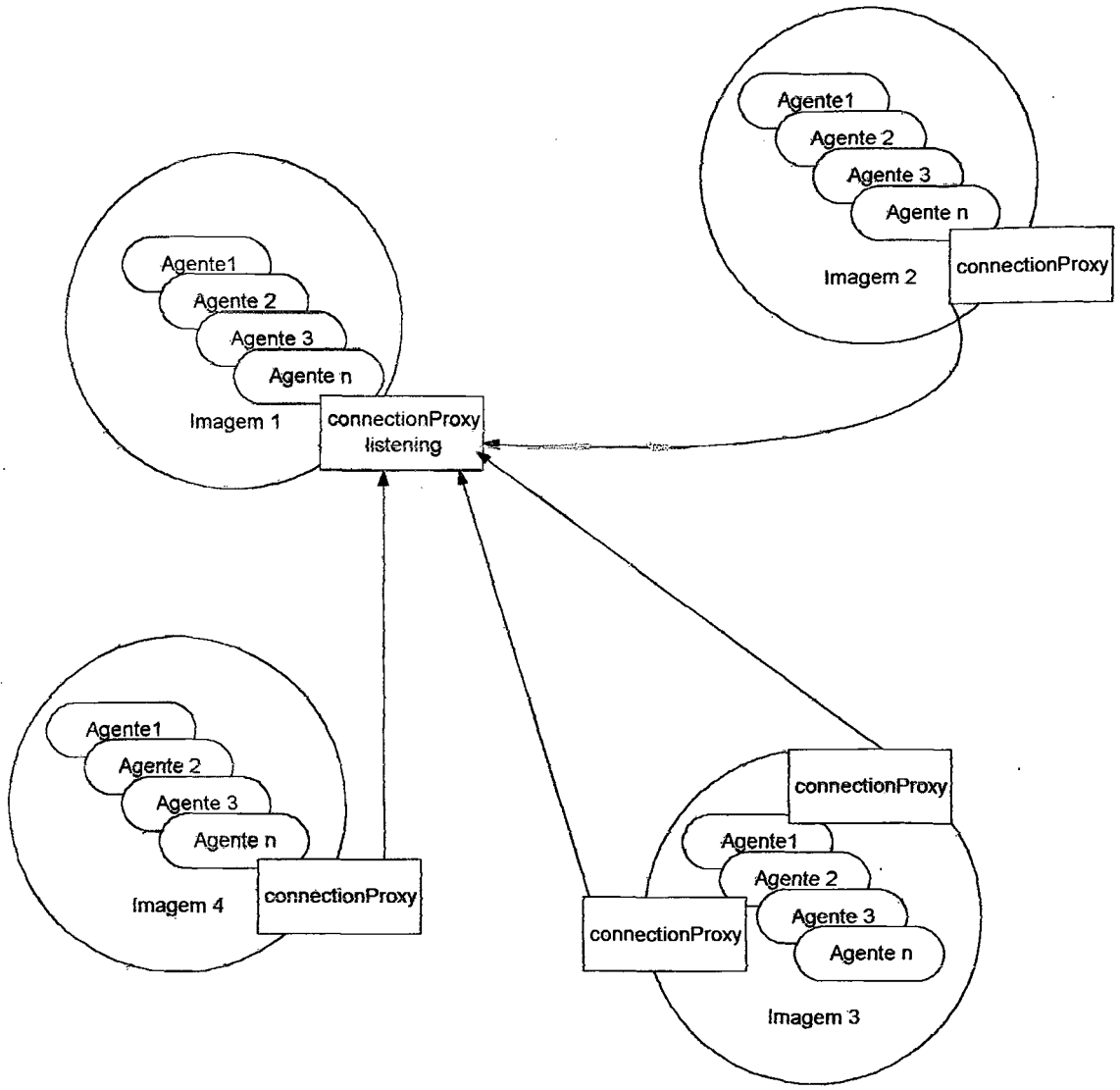


Figura 4: Topologia centralizada

Uma variação desta topologia é a colocação de mais de uma ConnectionProxy como servidor na imagem centralizadora. Desta forma, seria ouvida mais de uma porta. Este tipo de topologia facilitaria a distribuição de carga, já que cada ConnectionProxy faz o seu redirecionamento.

Esta variação poderia ser útil para a criação de facilitadores. A figura 5 mostra um diagrama desta implementação.

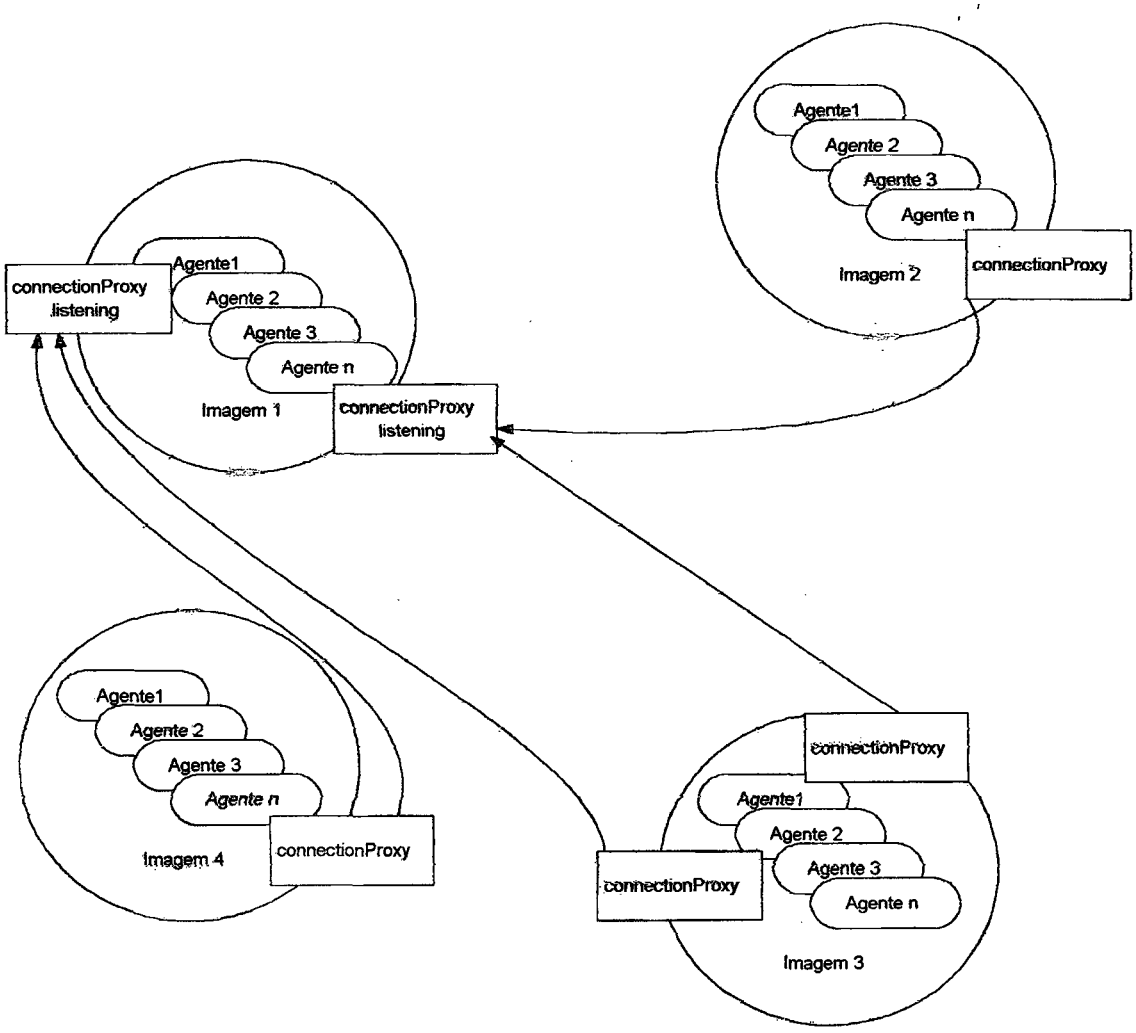


Figura 5: Variação da topologia centralizada.

5.2 Distribuído

Na topologia distribuída, todas ou quase todas as imagens podem aceitar conexões e fazer conexões, isto é, todas são clientes e servidores.

Este tipo de estrutura é útil quando não existe um agente ou conjunto de agentes caracteristicamente centralizadores.

Possíveis utilizações desta estrutura são sistemas produtor X intermediário X consumidor e colônias de agentes.

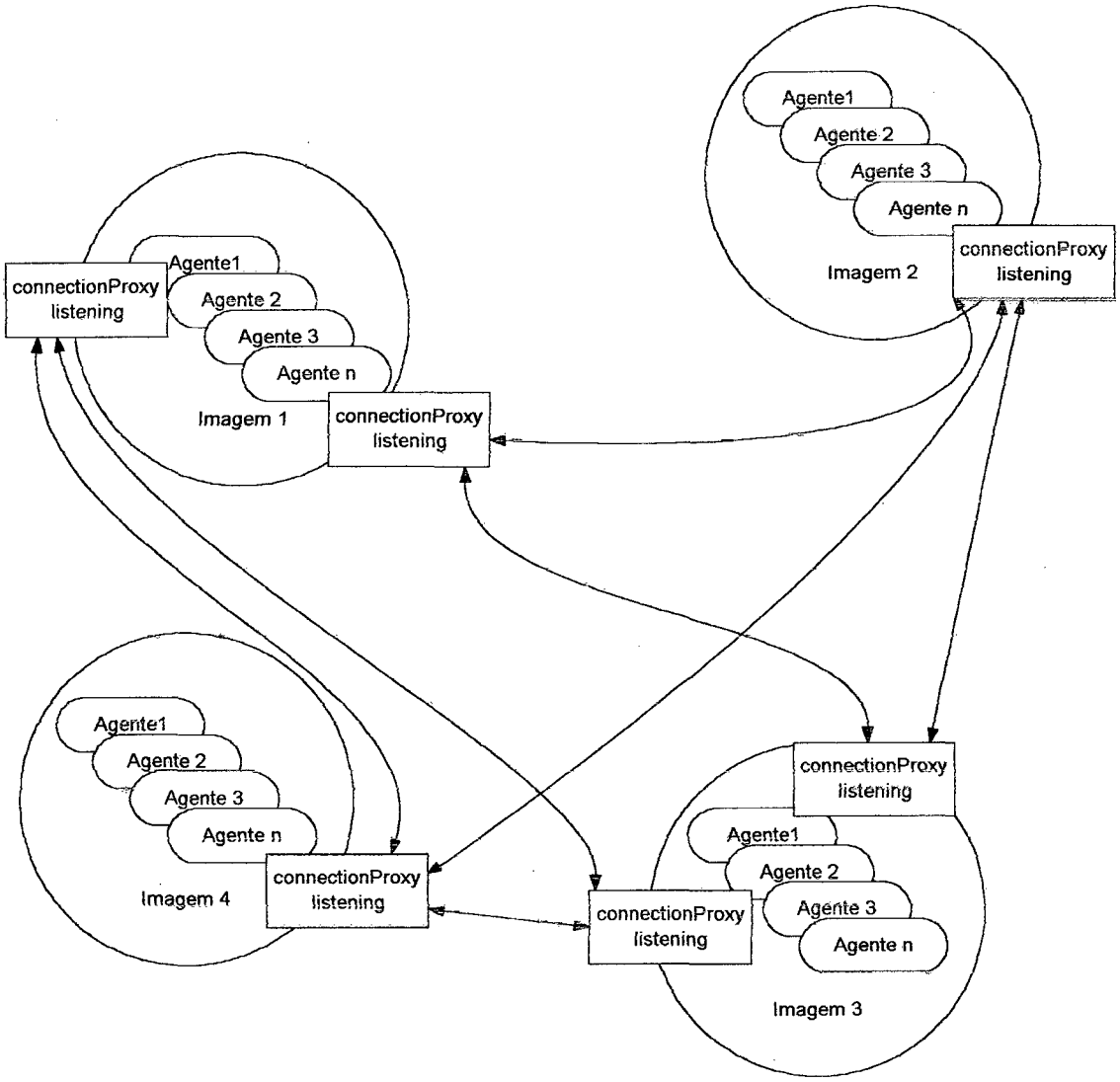


Figura 6: Topologia distribuída.

6 Exemplo de Implementação

Apenas para exemplificar a utilização da base de classes aqui definida, será apresentado um exemplo de implementação da comunicação entre agentes em um mercado artificial de ações. A estrutura de um mercado artificial de ações é bastante simples. Existem agentes negociadores e o agente mercado. Todos os agentes negociadores conhecem o agente mercado e o agente mercado conhece todos os agentes negociadores.

Existem três tipos de mensagens possíveis entre os agentes negociadores e o agente mercado:

Informação: são mensagens que solicitam informações sobre uma ação específica, sobre todas as ações ou sobre as solicitações dos agentes negociadores.

Negociação: são mensagens que tratam da negociação de ações. Entre elas, as de solicitação de compra, venda e as de confirmação ou cancelamento.

Estes tipos de mensagens serão descritos em uma linguagem que chamaremos de Linguagem de Manipulação de Ações ou *LMA*, como usaremos na cláusula *:language*.

6.1 Definição da LMA

Sintaxe:

Operação; Argumento1; Argumento2; ...; ArgumentoN

Mensagens:

Mensagem	Ação
cotacao; **codigoAcao**	Solicita informações sobre a cotação de uma ação específica. Retorna uma cadeia de caracteres com a seguinte configuração: codigoAcao ultNegociação ultCotacao ultQuantidade qtOfCompra vlOfCompra qtOfVenda vlOfCompra. Em :Language, teremos Cotação
cotacoes	Solicita informações sobre a cotação de todas as ações em negociação na bolsa de ações. Retorna uma cadeia de caracteres com a seguinte configuração: codigoAcao ultNegociação ultCotacao ultQuantidade qtOfCompra vlOfCompra qtOfVenda vlOfCompra Em :Language, teremos Cotacao
resp; codigoAcao; ultNegociação; ultCotacao; ultQuantidade; qtOfCompra; vlOfCompra; qtOfVenda; vlOfCompra	Resposta à cotação.
Oferta; **codigoAcao; tipo;**	Solicita informações sobre a oferta do **tipo** da ação representada no **codigoAcao**. O tipo pode ser *venda* ou *compra*.
Negociação	
svenda; **codigoAcao; quantidade; valor**	Solicita a inclusão de uma oferta de venda na fila de ofertas para a ação.
scompra; **codigoAcao; quantidade; valor**	Solicita a inclusão de uma oferta de compra na fila de ofertas para a ação.
ccompra; **codigoAcao; quantidade; valor**	confirma a operação de compra.
cvenda; **codigoAcao; quantidade; valor**	confirma a operação de venda.

Tabela 3: Mensagens definidas na LMA

A ontologia usada é *BolsaArtificial*.

Os diálogos possíveis são os seguintes:

Solicitar uma cotação:

(ask-one

:sender *codNegociador*

:content *"cotacao; codAcao"*

:receiver *mercadoArtificial*

:reply-with *codNegociador+n*

:language *LMA*

:ontology *BolsaArtificial*)

A mensagem de resposta, caso o código da ação exista, é a seguinte:

(tell

:sender mercadoArtificial

:content "RESP; codigoAcao; ultNegociação; ultCotacao; ultQuantidade; qtOfCompra; vlOfCompra; qtOfVenda; vlOfCompra"

:receiver codNegociador

:in-reply-to codNegociador+n

:language LMA

:ontology BolsaArtificial)

Caso o código da ação não exista, a resposta é a que segue:

(deny

:sender mercadoArtificial

:content cotacao; codAcao

:receiver codNegociador

:in-reply-to codNegociador+n

:reply-with n

:language LMA

:ontology BolsaArtificial)

Solicitar todas as cotações

(ask-all

:sender codNegociador

:content "cotacoes"

:receiver mercadoArtificial

:reply-with codNegociador+n

:language LMA

:ontology BolsaArtificial)

A mensagem de resposta, caso o código da ação exista, é a seguinte:

(tell

:sender mercadoArtificial

:content "RESP; codigoAcao1; ultNegociação; ultCotacao; ultQuantidade; qtOfCompra; vlOfCompra; qtOfVenda; vlOfCompra; RESP; codigoAcao2; ultNegociação; ultCotacao; ultQuantidade; qtOfCompra; vlOfCompra; qtOfVenda; vlOfCompra;...;RESP; codigoAcaoN; ultNegociação; ultCotacao; ultQuantidade; qtOfCompra; vlOfCompra; qtOfVenda; vlOfCompra "

:receiver codNegociador

:in-reply-to codNegociador+n

:reply-with n

:language LMA

:ontology BolsaArtificial)

Solicitar as ofertas do negociador para uma determinada ação:

(ask-all

:sender codNegociador

:content "oferta; codAcao; tipo"
:receiver mercadoArtificial
:reply-with codNegociador+n
:language LMA
:ontology BolsaArtificial)

Respostas:

(tell
:sender mercadoArtificial
:content " lista de todas as ofertas separadas por <CR> "
:receiver codNegociador
:in-reply-to codNegociador+n
:reply-with n
:language LMA
:ontology BolsaArtificial)
(deny
:sender mercadoArtificial
:content "oferta; codAcao; tipo"
:receiver codNegociador
:in-reply-to codNegociador+n
:reply-with n
:language LMA
:ontology BolsaArtificial)

Solicitar a inclusão de uma oferta de compra ou venda:

(insert
:sender codNegociador
:content "scompra; codigoAcao; quantidade; valor" ou
:content "svenda; codigoAcao; quantidade; valor"
:receiver mercadoArtificial

:reply-with codNegociador+n
:language LMA
:ontology BolsaArtificial)

Solicitar a confirmação de uma operação de compra ou venda:

(achive
:sender mercadoArtificial
:content "ccompra; codigoAcao; quantidade; valor" ou
:content "cvenda; codigoAcao; quantidade; valor"
:receiver codNegociador
:reply-with n
:language LMA
:ontology BolsaArtificial)

Como resposta positiva:

(tell
:sender codNegociador
:content "ccompra; codigoAcao; quantidade; valor" ou
:content "cvenda; codigoAcao; quantidade; valor"
:receiver mercadoArtificial
:in-reply-to n
:reply-with codNegociador+n
:language LMA
:ontology BolsaArtificial)

Como resposta negativa:

(deny
:sender codNegociador
:content "ccompra; codigoAcao; quantidade; valor" ou
:content "cvenda; codigoAcao; quantidade; valor"

:receiver codNegociador
:in-reply-to n
:reply-with codNegociador+n
:language LMA
:ontology BolsaArtificial)

6.2 Implementação

Não será examinada aqui a implementação completa, mas apenas o conjunto de códigos necessários para suprir o mecanismo de comunicação via KQML.

6.2.1 Instanciação do KqmlTreatment e configuração

No agente negociador:

Instancia o tratador de mensagens KQML.

```
kqmlTreatment :=KqmlTreatment new: self.
```

Adiciona a ontologia 'BolsaArtificial'

```
kqmlTreatment addOntology: 'BolsaArtificial'.
```

Adiciona a linguagem 'LMA'

```
kqmlTreatment addLanguage: 'LMA'.
```

Adiciona o agente '*MercadoArtificial*' na lista de agentes conhecidos.

```
kqmlTreatment addAgents: 'MercadoArtificial'.
```

Informa ao tratador o que fazer com as seguintes *performatives*. Caso não seja informado, o padrão é *#sorry* para as *performatives* padrão.

```
kqmlTreatment addSemantical: 'tell' semantical: #agent.
```

O agente terá que implementar um método com o seguinte protocolo:

```
tell: aMessageKqml
```

Que será executado automaticamente quando uma mensagem com *performative tell* for recebida.

```
kqmlTreatment addSemantical: 'deny' semantical: #agent.
kqmlTreatment addSemantical: 'achive' semantical: #agent.
```

No agente mercado:

Instancia o tratador de mensagens KQML

```
kqmlTreatment :=lTreatment new: self.
```

Adiciona a ontologia '*BolsaArtificial*'

```
kqmlTreatment addOntology: 'BolsaArtificial'.
```

Adiciona a linguagem 'LMA'

```
kqmlTreatment addLanguage: 'LMA'.
```

Informa ao tratador o que fazer com as seguintes *performatives*. Caso não seja informado, o padrão é *#sorry* para as *performatives* padrão.

```
kqmlTreatment addSemantical: 'ask-if' semantical: #agent.
kqmlTreatment addSemantical: 'ask-all' semantical: #agent.
kqmlTreatment addSemantical: 'insert' semantical: #agent.
kqmlTreatment addSemantical: 'delete-one' semantical: #agent.
kqmlTreatment addSemantical: 'tell' semantical: #agent.
kqmlTreatment addSemantical: 'deny' semantical: #agent.
```

6.2.2 Utilizando o ConnectionProxy

A classe *ConnectionProxy* que fornece o meio de comunicação entre agentes.

Para prover o envio e recebimento de mensagens, existem dois métodos principais:

1. **getMessageQueue**: *agentName* - Pega, se disponível, uma fila de mensagens (*SharedQueue*) do meio para o agente. Caso exista, esta será devolvida para o agente, já com o parser feito de todas as mensagens na fila. O retorno da fila em vez do de uma única mensagem é válido quando o agente pretende um maior controle sobre o recebimento de mensagens.
2. **sendMessage**: *aKqmlMessage.* – Coloca uma mensagem no meio e cuida do seu roteamento.

Além desses métodos, existem outros também de grande importância, cujos usos vamos exemplificar.

É importante reparar que aqui vai acontecer diferença entre *ConnectionProxy* apenas clientes e *ConnectionProxy* que também sejam servidores.

Primeiro demonstrar-se-á uma *ConnectionProxy* no modo Servidor. No exemplo, isto ocorre na imagem onde estava a classe *BolsaDeValores.*

Primeiramente veremos a instanciação e configuração. Este código está inserido no método *initializeNetwork.*

```
connectionProxy :=ConectionProxy new.
```

Aqui é criada uma nova instância da *ConnectionProxy.*

```
connectionProxy portNumber: 8555.
```

A definição do *portNumber* igual a 8555 significa que esta *ConnectionProxy* está em modo servidor. Caso não fosse descrita uma porta ou esta fosse declarada como *nil*, esta *ConnectionProxy* não poderia receber conexões, apenas realizá-las.

```
connectionProxy initialize
```

Cria dois processos: um fica ouvindo a porta definida e disparando *sockets* novos para cada conexão nova. O outro processo fica verificando as conexões já realizadas.

No caso de não estar definida uma porta, apenas o segundo processo será criado.

```
self name: 'BVUFSC'
```

Este método não pertence à definição do *connectionProxy*, está aqui apenas por conveniência, especificando o nome do agente. É importante que isto seja feito antes do código que segue, não importando aonde.

```
connectionProxy localAgents: self name agent: self.
```

Adiciona um agente local ao *connectionProxy*. No caso, apenas o agente *BVUFSC* foi adicionado. Isto é necessário para que o *connectionProxy* possa rotear as mensagens endereçadas para esse agente.

No caso de *connectionProxy* apenas cliente, no exemplo, o negociador faz esse papel. O conteúdo a seguir é parte do método *initializeNetwork*.

```
connectionProxy := ConnectionProxy new.
```

Aqui é criada uma nova instância da *ConnectionProxy*.

```
connectionProxy initialize
```

Como não foi definida uma porta, é criado apenas um processo que fica verificando todos os *sockets* ativos.

Do lado bolsa de valores, o *connectionProxy* fica apenas esperando novas conexões e, no fim do ciclo de vida da imagem, ou do agente, a finalização dos processos.

O agente bolsa de valores não faz conexões, apenas recebe. O que já não acontece com o agente negociador.

Para um agente poder entrar em contato com outro, ele necessita primeiramente abrir um canal de conexão para um agente.

```
connectionProxy connect: 'BVUFSC' address: stockAddress port: stockPort.
```

Este método cria um *socket* para uma agente, no caso *BVUFSC* localizado em uma *connectionProxy* especificada. Isto não significa que o agente que está chamando a *connectionProxy* já esteja registrado na *connectionProxy*. O modelo prevê que toda vez que um novo agente for conectado, mesmo quando este estiver numa mesma *connectionProxy*, será criada uma nova *socket*[26]. Isto melhora consideravelmente a performance do processo.

O registro ainda é necessário, pelo menos no nível da *connectionProxy*, em razão do problema da nomeação do agente. O agente ainda não conhece o nome do agente e, ainda, o agente pode já ter outro agente registrado com o mesmo nome, o que não pode acontecer.

---

[26] Um socket é uma estrutura que controla a conexão entre duas portas TCP/IP. Cada porta pode ter apenas uma conexão ou socket.

Um grande problema deste modelo é a existência de agentes pertencentes a mais de uma sociedade. Existe uma grande probabilidade de o nome do agente em questão já estar sendo usado na segunda sociedade. Desta forma, o agente terá que mudar seu nome na primeira sociedade, ou possuir dois nomes ou denotações, os quais serão utilizados de maneiras distintas pelos agentes das sociedades. Como será colocado mais adiante, uma solução é a utilização de um facilitador ou servidor de nomes.

Abaixo, uma forma de realizar o "handshake" de modo a conseguir delegar um nome para o agente e realizar o registro:

```
[registred] whileFalse: [
(Delay forMilliseconds: 50) wait.
   self nome: (self nameFormation: (self nameFormation: '')).
  connectionProxy localAgents: self name agent: self.
  kqmlMessage := KqmlMessage new.
  kqmlMessage  performative: 'register';
                    sender: self name;
                    receiver: 'BVUFSC'.
  self sendMessage: kqmlMessage.
```

Esta primeira parte do código cuida de verificar se o registro já foi realizado (variável de instância *registred*); caso não, é gerado um nome, registrado como agente local, e enviada uma mensagem KQML "*register*".

O Primeiro *Delay*, aparentemente inútil, é importante para evitar o overhead do envio de consecutivas mensagens de conexão. Além disso, pelo excesso de tráfego em uma rede, pode haver demora na conexão. Na seção abaixo, o *delay* serve para esperar o recebimento e tratamento da mensagem pelo receptor.

```
(Delay forMilliseconds: 200) wait.
[(connectionProxy haveMessage: self name)|(registred)]        whileFalse: [
        (Delay forMilliseconds: 500) wait.
].
```

Nesta parte do código, o agente fica esperando por uma mensagem. Quando esta for recebida, o tratamento será despachado pelo método *receiveMessage*, dependendo da mensagem. Caso seja *OK*, ou *REGISTER*, o agente recebe o nome retornado na mensagem e termina o processo de registro. Caso contrário, por uma mensagem *UNREGISTER*, continua o processo até que um nome seja aceito.

```
        self receiveMessage.
            (Delay forMilliseconds: 3000) wait.
            self receiveMessage.
          (Delay forMilliseconds: 200) wait.
    ].
```

Após o registro, o agente negociador solicita ao agente *BVUFSC* a cotação de todas as ações em negociação. Este procedimento atualiza a base de conhecimento do agente negociador.

```
[(connectionProxy haveMessage: self name)] whileFalse: [
        (Delay forMilliseconds: 500) wait.
].
self receiveMessage.
```

Outra parte do código que merece ser observada é a implementação do método receiveMessage.

```
(connectionProxy haveMessage: self name) ifTrue: [
kqmlMessage_connectionProxy getMessageQueue: self name.
(kqmlMessage isNil) ifFalse: [
  kqmlTreatment evaluateMessage: kqmlMessage.
  ^true.
] ifTrue: [
^false.
].
].
```

Este método verifica se existe alguma mensagem; se houver, ele vai retirar a primeira e mandar executá-la pelo *KqmlTreatment*.

Após o processo de inicialização, este método será executado continuamente até o fim da vida do agente.

6.3 Análise dos Resultados

De um modo geral, os agentes se comportaram bem e a comunicação funcionou a contento, sem maiores problemas. As quedas de conexões foram satisfatoriamente contornadas pelo sistema, sem que o comportamento fosse duramente afetado.

O agente BVUFSC conseguiu tratar até 120 mensagens por segundo, tendo a ele conectados cerca de 80 agentes negociadores.

O tempo máximo gasto no processo de handshake (designação de nomes) foi de 5 (cinco) minutos. O meio físico foi uma rede ethernet 10 Mbits. Os computadores utilizados foram uma RS6000 IBM com processar powerPC 603 de 233 Mhz, onde foi colocado o *BVUFSC*. E os agentes foram executados em 5 (cinco) máquinas, variando de AMD K6-2 a Pentium II.

A maior carga do agente *BVUFSC* era o retorno da solicitação de cotação de todas as ações, que gerava uma mensagem com 64 KB. Esta mensagem era gerada aproximadamente uma vez por segundo para cada agente registrado.

O maior *throughtput* alcançado foi de 253 KB/s, velocidade excelente, quando analisado o custo de processamento das mensagens.

Como os agentes tinham comportamento bastante simples, as simulações na bolsa apresentaram-se estáveis, não apresentando comportamento complexo. Porém ficou claro que este ambiente é perfeito para esse tipo de simulação - observando-se o que será abordado no capítulo 7- pois permite um máximo poder de processamento por agente.

7 Vantagens e Problemas da Implementação

A principal vantagem da interface é a sua leveza e simplicidade. Cada imagem pode conter um conjunto de agentes compartilhando a mesma *connectionProxy*, evitando assim a existência de uma interface de rede para cada agente. Este tipo de estrutura também diminuiu o *overhead* da comunicação entre agentes na mesma imagem, já que a mensagem não necessita ir para rede. Outra grande vantagem deste tipo de implementação é a identificação automática da localização do agente e o controle da conexão.

Como problemas, é possível identificar:

1. Falta de um mecanismo de busca de outras *connectionProxy* (imagens[27]). A implementação necessita que cada agente conheça a imagem em que quer se conectar. Uma forma de contornar este problema é a implementação de agentes facilitadores, onde todos os outros agentes se registram e que manteriam um cadastro de todos os agentes e seus endereços. Exemplo de funcionalidades possíveis para facilitadores utilizando KQML pode ser visto no artigo de Finin [FIN95]. A grande dificuldade de implementação de um mecanismo de busca puro é a quantidade extremamente grande de endereços possíveis e a inexistência de portas pandrões (também chamado de serviços TCP/IP), o que tornaria a busca um problema computacionalmente complexo.

---

[27] Uma imagem aqui pode ser considerada como um ambiente smalltalk (de um modo geral, será apenas um processo de S.O.)

2. Falta de um mecanismo de controle de nomes de agentes. Este problema é semelhante ao anterior. Na implementação proposta, o nome do agente é baseado no relacionamento entre duas imagens, isto é, um agente só pode ser registrado em uma imagem se o nome por ele solicitado não estiver em uso em uma determinada imagem. Este tipo de estrutura pode trazer grandes problemas em estruturas distribuídas, quando um agente pode estar ligado a várias outras imagens, as quais, por sua vez, podem ter muitas outras ligações a outras imagens, sendo provável assim a utilização do mesmo nome. Da mesma forma que o problema anterior, uma solução simples seria a utilização de um facilitador para registrar os agentes e seus nomes, além de manter uma base de nomes utilizados. Críticas a este tipo de estrutura centralizada poderão ser contornadas com a utilização de uma rede de facilitadores permitindo o balanceamento dos recursos. Ainda como solução alternativa, pode-se utilizar nomes com referência à localização do agente (imagem, endereço e porta). Este tipo de solução é ideal quando os agentes não possuem mobilidade.

8 Uso de Sistemas Multiagentes em Simulações

A utilização de sistemas multiagentes é, aparentemente, a melhor opção quando se tem uma grande quantidade de objetos ou ítens em uma simulação, principalmente quando é visível a possibilidade de tradução de um objeto, item ou regra para um agente. Neste momento, é necessária a atenção para alguns detalhes:

1. Quanto à forma de comunicação: síncrona/assíncrona.
2. Quanto ao modelo da simulação: discreta ou contínua.
3. Quanto à questão tempo: real, simulado, único ou independente.
4. Quanto ao disparo e fim da execução e coordenação da simulação.
5. Quanto à forma de inquisição de resultados e sua compilação.

Estes detalhes podem, caso não estudados, inviabilizar uma simulação usando sistemas multiagentes.

Os ítens 1, 2 e 4, comunicação, modelo e controle, são parcialmente resolvidos utilizando a própria linguagem KQML. [FIN95][JAT99][BUR95]

A línguagem KQML tem funções de coordenação que podem ser facilmente implementadas pelos agentes. Porém, o problema de cordenação ainda tem muito a ser estudado.

De um modo geral, uma simulação tem pontos de início e parada bem definidos. Isto não é problema quando não existe a distribuição de agentes em várias maquinas.

O disparo, que inclui a criação e inicialização do agente, quando local, é algo extremamente simples. O problema aparece quando o agente é remoto.

No caso de um agente remoto, pode-se citar as seguintes possibilidades:

- O agente já está ativo em modo de espera, aguardando que o agente controlador da simulação dispare a simulação;
- O agente não foi disparado e será necessária a intervenção local;
- A existência de um mecanismo permitindo a mobilidade do agente, por exemplo, uma estrutura que aguarda o envio do agente e o seu disparo de maneira remota. Neste caso, seria bastante interessante que este mecanismo permitisse o controle da simulação, isto é, sincronismo, cronômetro ou agenda, e intervenções na execução.

De um modo geral, a última possibilidade parece mais interessante. Porém, este tipo de solução tem algumas restrições. Entre elas, a dependência ou do S.O. ou de uma arquitetura de agentes. Para esse modelo, ainda existe o problema com a segurança e autenticação. Itens que se não observados, podem permitir a criação de agentes mal intencionados[28], os quais poderiam infectar ou degenerar a sociedade.

Aceitando-se a restrição da limitação da arquitetura, pode-se construir uma arquitetura de controle e despacho bastante interessante.

Uma sugestão para essa arquitetura é a seguinte:

---

[28] Algo semelhante aos vírus e cavalos de Tróia que comumente invadem sistemas operacionais e aplicativos com baixo nível de segurança.

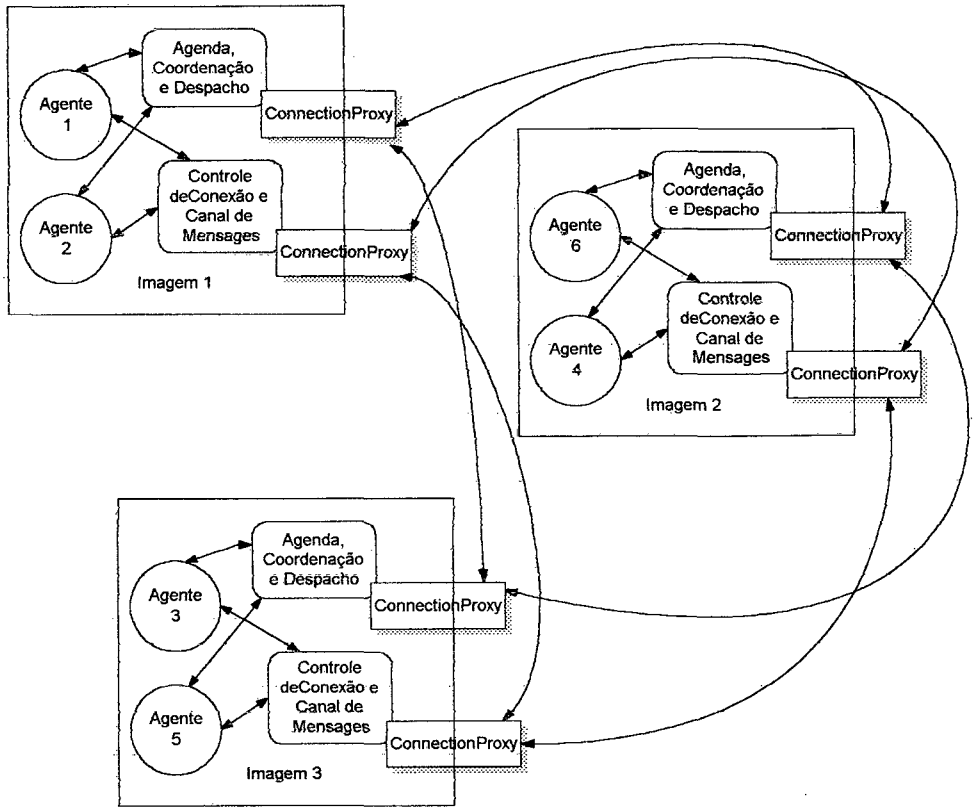


Figura 7-Arquitetura Proposta

A sugestão é uma arquitetura baseada em dois serviços básicos: *Comunicação* e *Controle*.

O serviço de comunicação é basicamente um canal para trânsito de mensagens normais.

O serviço de controle é o responsável pelo despacho, agendamento e coordenação. Como cada imagem pode ter mais de uma sociedade ou simulação, o despacho, a agenda e a coordenação devem ser individualizados por sociedade ou simulação. Partindo dessa idéia, deve ficar claro que um agente seria restrito a uma única sociedade ou simulação.

Os serviços, por sua vez, podem ser implementados como agentes. Porém, é extremamente aconselhável que a comunicação interna seja feita de forma direta, de modo a evitar o *overhead*[29] do KQML.

Este modelo permite que qualquer imagem tenha a função de disparadora, e que até mais de uma imagem faça o despacho. O modo mais simples de realizar o despacho é enviar código smalltalk, sendo este compilado e executado na imagem receptora.

Não é escopo deste trabalho a discussão mais aprofundada deste tema, tampouco a implementação destes mecanismos. Porém, esta discussão é válida pelos problemas enfrentados na implementação e execução do problema exemplo.

Esta dificuldade não existe em sistemas multiagentes como o SWARM [LAB94]. O SWARM não implementa KQML e também não é executado de forma distribuída. Porém, permite uma grande facilidade na construção de agentes, principalmente quando estes forem utilizados em simulações. Isto ocorre porque todos os agentes estão locais e são controlados pelo fluxo de programação do ambiente. Pela definição de sistemas multiagentes e da inteligência artificial distribuída, este modelo não atende a todos os quesitos de sua definição, já que não existe uma distribuição real, apenas uma distribuição em processos compartilhando o mesmo processador.

---

[29] Carga adicional de processamento

9 Outras Implementações

Uma das implementações mais populares para sistemas multiagentes, o *JATLite*[JAT99] é uma referência a ser considerada. O *SWARM* também é um ótimo exemplo de sistemas multiagentes, porem não implementa nem KQML, nem comunicação via rede.

O *JATLite* implementa, além da linguagem *KQML*, todo um ambiente de construção de sistemas multiagentes. O *JATLite* é escrito em *JAVA* e tem a estrutura abaixo:

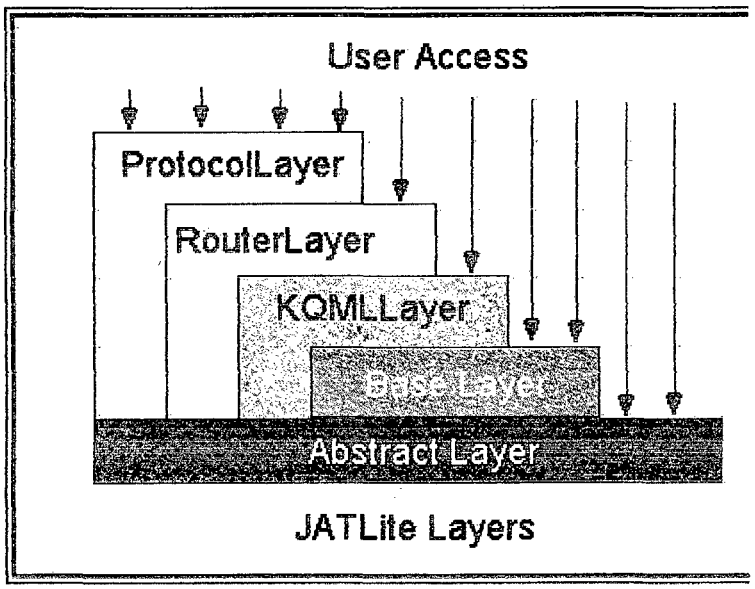


Figura 4: Estrutura das camadas do *JATLite*[JAT99]

A *Abstract Layer* (camada abstrata) provê uma coleção de classes abstratas necessárias à implementação do *JATLite*.

A *Base Layer* (camada de base) provê a comunicação baseada em TCP/IP e na abstract layer. Nesta camada não há restrição do tipo de mensagem.

A *KQML Layer* (camada KQML) provê o armazenamento e o parsing das mensagens KQML.

A *Router Layer* (camada de roteamento) provê o registro de nome, roteamento de mensagens e enfileiramento. Todos os agentes recebem e mandam mensagens via Router.

A *Protocol Layer* (camada de protocolo) provê serviços de protocolos, como por exemplo *SMTP* ou *FTP*.

O *JATLite* não provê um tratamento mais aprimorado da semântica das mensagens KQML, e ainda obriga o agente a aceitar a política de fila do mesmo.

Um grande problema do *JATLite* é a necessidade de incorporar as subcamadas, isto é, quando se quer usar a *KQML layer*, é obrigatória a utilização da Base *Layer* e Abstract *Layer*. Isto pode tornar o agente pesado e amarrá-lo demasiadamente.

A estrutura apresentada aqui tem a forma de componentes que podem ser agregados a qualquer software, ou agente de software. É independente da implementação das demais estruturas do agente. A política de fila, bem como a estrutura de comunicação em rede utilizada, fica totalmente a cargo do usuário ou do ambiente de construção de sistemas multiagentes no qual está sendo criado o agente.

Comparando alguns ítens do *JATLite* e da implementação aqui apresentada, temos:

ITEM	*JATLite*	*Implementação*
LINGUAGEM	JAVA	SQUEAK SMALLTALK
PARSER	SIM	SIM
TRATAMENTO SEMÂNTICO	NÃO	SIM (Apenas um tratamento básico onde os comportamentos possíveis são: Responder com ERROR, Reponder com SORRY, ou executar tratamento definido no agente.)
CONTROLE DAS MENSAGENS	SIM	NÃO IMPLEMENTADO
TRATAMENTO DE REDE	SIM	SIM

Tabela 4: Comparação entre o JATLite e a Implementação aqui sugerida.

10 Conclusão e Trabalhos Futuros

10.1 Conclusão

O KQML desponta como uma linguagem padrão de comunicação entre agentes. A sua utilização é quase obrigatória quando se trata de comunicação entre agentes ou dentro de sistemas multiagentes.

Este trabalho consegue modelar uma estrutura que facilita o tratamento das mensagens em KQML, realizando o seu "parsing" e auxiliando no tratamento semântico.

Em relação ao JATLite[JAT99], e especificamente ao KQML Layer, a estrutura da implementação aqui apresentada tem a vantagem de permitir maior controle do agente sobre as mensagens, seu enfileiramento, escalonamento e forma de comunicação. Este fato permite que seja criado um agente muito leve do ponto de vista computacional.

A construção da classe ConnectionProxy, que realiza o controle de conexão, o registro e o envio e recebimento de mensagens, vem facilitar a construção de sistemas multiagentes sobre uma rede TCP/IP, já que realiza o roteamento das mesmas.

A construção da Bolsa de Valores Artificial permitiu detectar vários problemas na construção de sistemas multiagentes, realmente distribuídos, para uso em simulações. A partir dessas dificuldades, foram apresentadas sugestões para a solução

do problema, que deverão ser estudadas e discutidas em outro momento, pois este tipo de aplicação apresenta uma grande complexidade e modelos a serem analisados.

A especificação está de um modo geral consistente com o que foi definido no início deste trabalho, que atinge assim o seu objetivo.

10.2 Trabalhos Futuros

Como trabalhos futuros, é importante a construção de agentes facilitadores, para o registro de nome e para roteamento, e o estudo do problema do controle para o uso de sistemas multiagentes em simulação.

A construção de agentes facilitadores vem da necessidade de mecanismos mais adequados, que venham a prover uma maior velocidade na negociação do problema de nomeação de agentes. Este tipo de agente funcionaria como um cartório virtual, onde cada agente, no momento em que fosse criado, tentaria registrar um nome, podendo antes consultar os nomes já registrados. Este tipo de agente também teria como função registrar o óbito de um agente, liberando assim o seu nome.

Um facilitador de nome, ou cartório virtual, também deveria prover métodos para realizar a autenticação dos agentes, simulando o serviço de cartório real e permitindo transações seguras entre agentes.

O problema do controle para uso de sistemas multiagentes em simulação aparece de forma insignificante, já que existe a crença na disponibilização dos mecanismos de controle do próprio KQML. Porém, deve ficar claro que o KQML tem controle apenas

no fluxo e semântica das mensagens, e o controle necessário, neste caso, é o do fluxo de execução, disparo e parada de execução dos agentes, além da possibilidade de verificação dos resultados.

É bastante interessante um estudo aprofundado do mecanismo proposto e a apresentação de outros mecanismos para o tratamento do problema do controle de execução.

Ainda como trabalhos futuros, vislumbra-se a criação de agentes mais inteligentes para executar na bolsa artificial de valores e a melhoria do ambiente de construção para os agentes negociadores.

11 Bibliografia

11.1 Referencias

[FIN95] FININ, Tim; LABROU, Yannis e MAYFIELD, James. **KQML as an agent comunication language.** University of Maryland Baltimore County. Setembro, 1995.

[JEN96] JENNINGS, Nick e WOOLDRIDGE, Michael . **Software Agents.** in IEEE Review, janeiro de 1996, pp 17-20.

[FRA96] FRANKLIN, Stan e GRAESSER, Art. **Is it na Agent, or just a Program?: A Taxonomy for Autonomous Agents.** Proceedings of the Third Workshop on Agent Theories, Architectures, and Languages, Springer-Verlag, 1996.

[FIN93] FININ, Tim et al. **DRAFT: Specification of the KQML Agent-Communication Language.** The DARPA Knowledge Sharing Initiative. External Interfaces Working Group, junho 1993.

[FIP97] **FIPA '97 Specification Part 2, Agent Communication Language.** http://209.61.157.155/specs/fipa00003/OC00003.doc - 01/09/2000.

[JAT99] **JAT-Lite.**http://cdr.stanford.edu/ProcessLink/papers/JATL.html. - 06/10/1999.

[RIV99] RIVERO, Sérgio. **Um Framework para Simulação Econômica Baseada em um Modelo de Agente Adaptativo Antecipatório com Racionalidade Limitada.** Dissertação de Mestrado EPS/UFSC, Agosto 1999

[SEA69] SEARLE, John R. **Speech acts: an essay in the philosophy of language.** Cambridge. Univ. Press. 1969.

[LAB94] LABROU, Yannis e FININ, Tim. **A Semantis Approach for KQML - A General Purpose Communication Language for Agents**. Third International Conference on Information and Knowledge Management(CIKM'94), Novembro 1994.

[BUR95] BURMEISTER, Birgit e HADDADI, Afsaneh e SUNDERMEYER, Kurt. **Generic, Configurable, Cooperation Protocols for Multi-Agent Systems.** Lecture Notes in Artificial Intelligence - From Reaction To Cognition. 1995.

[OLI96] OLIVEIRA, Flavio M. **Inteligencia Artificial Distribuída.** IV Escoloa Regional de Informática, Londrina-Itajaí-Canoas. 1996.

[RUM91] RUMBAUGH, James. **Object-Oriented Modeling and Design.** New Jersey: Prentice Hall, 1991.

11.2 Bibliografia de Apoio

WOOLDRIDGE, Michael e JENNINGS, Nicholas R. **Intelligent Agents: Theory and Pratice.** ECAI-94 Workshop on Agent Theories, Architetures, and Languages. Berlin: Springer-Verlag. 1994.

THIRUNAVUKKARASU, Chelliah; FININ, Tim e MAYFIELD, James. **Secret Agents -- A Security Architecture for the KQML Agent Communication Language.** CIKM'95 Intelligent Information Agents Workshop, Baltimore, December 1995.

VIDAL, J.M., and DURFEE, E.H. **Building Agent Models in Economic Societies of Agents**: *AAAI-96 Workshop on Agent Modeling*, Portland, OR, July 1996.

ZENG, Dajun and SYCARA, Katia. **Benefits of Learning in Negotiation.** Proceedings of AAAI-97, 1997.

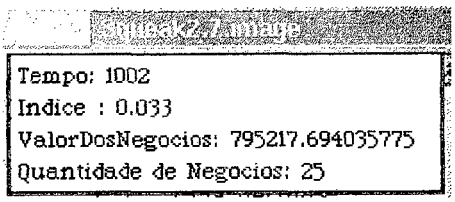
ARTHUR, W. Brian, et all. **Asset Pricing Under Endogeneous Expectations in na Artificial Stock Market.**

CHORAFAS, Dimitris N. **Agent Technology Handbook.** New York: McGraw-Hill, 1998.

GASPARI, Mauro. **Knowledge-Level Speech Acts.** Technical ReportUBLCS-97-3. Italy: University of Bologna, March,1997.

12 ANEXOS

12.1 Tela de Acompanhamento do Agente BVUFSC (Bolsa de Valores)



12.2 Log de Negociações do Agente BVUFSC

Hora Real; Tempo Simulado (S) ;Agentes
Ação; Valor
7:05:12 pm;387;matching;LR;HQ;
Tibras PNA;0.02682160706320498
7:05:16 pm;390;matching;II;IL;
Acesita PN;0.007389511620636565

7:05:21 pm;393;matching;LR;HQ;
Tibras PNA;0.043646223819224

7:05:39 pm;403;matching;LR;IL;
Acesita PN;0.00227830340306321

7:05:41 pm;404;matching;LR;II;
Tibras PNA;0.01178742187285665

7:05:46 pm;406;matching;IL;HQ;
Tibras PNA;0.043646223819224

7:05:47 pm;406;matching;II;KI;
Bahema PN;0.0639118065161167

12.3 Log de Indices do BVUFSC

```
Hora Real;Tempo Simulado(S);Indice;Valor C.H.;Negocios; Agentes em
Negocio
7:20:19 pm;1035;0.0031675566755318l3;2.596895331125927e7;28;9
7:20:23 pm;1036;0.0031675566755318l3;2.596895331125927e7;28;9
7:20:25 pm;1037;0.0031675566755318l3;2.596895331125927e7;28;9
7:20:27 pm;1038;0.00320320457716239;2.596896256835297e7;29;9
7:20:30 pm;1039;0.00320320457716239;2.596896256835297e7;29;9
7:20:32 pm;1040;0.00320320457716239;2.596896256835297e7;29;9
```

12.4 Log de Negociações do Agente KI (Negociador)

```
Inicio do log para o agente: KI
FORMATO: tempo - operacao - saldo - ativo - valores
7:05:36 pm;nil;Cotacoes;nil;
Tibras PNA;0.04364

7:05:37 pm;nil;Cotacoes;nil;Acesita ON;0.04364

7:05:37 pm;nil;Cotacoes;nil;
Bahema PN;0.04364
7:05:38 pm;3;solicitaCompra;1028422;
Sadia Concord PN;0.722176012911523

7:05:39 pm;5;solicitavenda;1028422;
Bahema PN;0.0503416842769515

7:05:39 pm;9;solicitavenda;1028422;
Bahema PN;0.05581019698025566

7:17:44 pm;3160;ConfirmaVenda;1.036677295093553e6;
Bahema PN;0.05370205754178036
```