4 T-Rex

T-Rex [51,44] é uma implementação do padrão arquitetural *Reflective Blackboard*, apresentado no capítulo anterior. Esta implementação, realizada na linguagem Java [26], é totalmente baseada na tecnologia de espaços de tuplas, mais especificamente no pacote IBM TSpaces [35]. Neste capítulo, apresentaremos em detalhes as principais características de T-Rex. T-Rex também pode ser encarado como um *framework* para o desenvolvimento de aplicações baseadas em agentes que necessitam da utilização de diferentes propriedades inter-agentes uma vez que ele oferece suporte à mobilidade, comunicação, persistência e coordenação dos agentes.

4.1 Visão Geral

4.1.1 Mapeamento do Padrão Reflective Blackboard

A implementação de T-Rex é baseada na variante Espaços de Tuplas Reflexivos (Seção 3.8) do padrão Reflective Blackboard. Em espaços de tuplas reflexivos, os meta-objetos são estruturados através de meta-tuplas, que encapsulam meta-informação e comportamento específico. Estas meta-tuplas são armazenadas em um espaço de tuplas denominado espaço de meta-nível ou meta-espaço. O espaço de meta-nível é ligado através do MOP ao blackboard do nível base, que por sua vez também é um espaço de tuplas, denominado espaço base. Desta forma, quando o MOP intercepta operações executadas no nível base ele busca por meta- informações no espaço de meta-nível.

As meta-tuplas em T-Rex são estruturadas como quádruplas do tipo (Rct, AgId, Op, T) que associam a reação (ou comportamento específico) Rct à execução de uma operação Op sobre a tupla T no espaço de nível base por parte do agente identificado por AgId. Uma vez que o meta-espaço é um espaço de tuplas, a busca por meta-tuplas também pode ser feita através de casamento de padrões. Conseqüentemente, uma meta-informação pode conter coringas, tornando-a mais genérica. Na verdade até mesmo a tupla T pode conter coringas (representados neste documento por *), ou outras tuplas. Por exemplo, a meta-tupla (Rct1, *, Write, (“Msg”, Ag1 , *)) associa a reação Rct1 à operação de escrita efetuada por qualquer agente sobre a tupla que representa uma mensagem endereçada ao agente identificado por Ag1, com um conteúdo qualquer. Em outras palavras, a presença desta tupla no espaço de meta-nível significa que a reação Rct1 será executada sobre qualquer mensagem enviada ao agente identificado por Ag1.

O protocolo de meta-objeto de T-Rex intercepta qualquer operação realizada no espaço de nível base e busca, através de uma operação de leitura no espaço de meta-nível, meta-tuplas associadas à operação executada. Caso exista alguma meta-tupla, a reação associada a ela é recuperada e executada, sendo retornado ao agente o resultado desta execução.

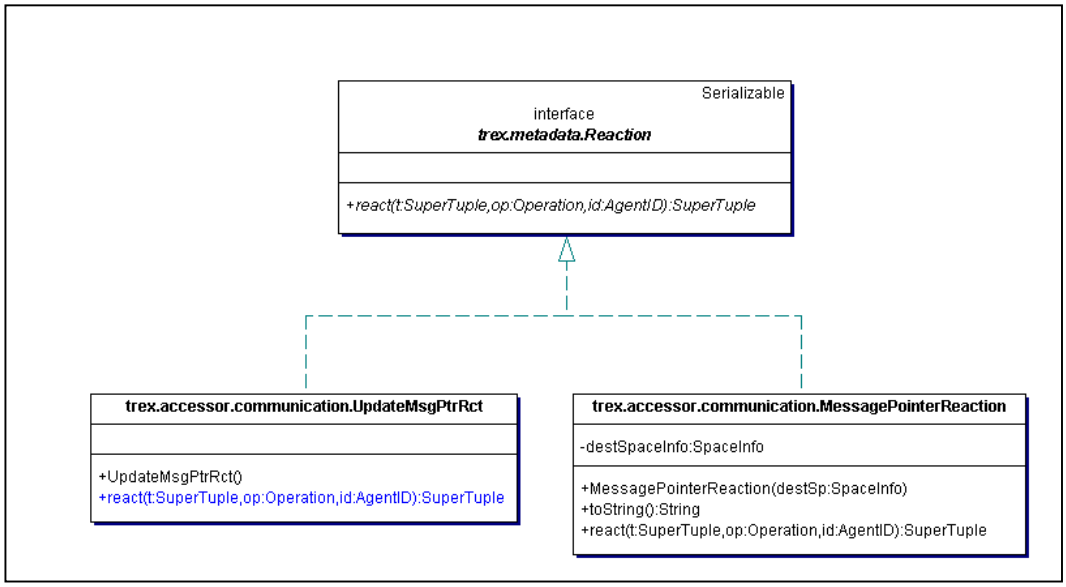

Figura 8 A interface Reaction e algumas de suas implementações

As reações em T-Rex são objetos que implementam a interface Reaction. Esta interface possui um método que especifica os parâmetros que devem ser levados em consideração na execução de uma reação. Estes parâmetros são o identificador do agente que executou a operação, a operação executada e a tupla que é objeto da operação no nível base. Estes parâmetros são na verdade a reificação [37] de dados existentes no nível base. As reações sempre retornam uma tupla que pode ser a própria tupla passada como parâmetro, esta tupla modificada ou uma nova tupla criada pela execução da reação. Em T-Rex, não há nenhuma restrição de acesso à execução das reações, podendo estas ser definidas de forma diferente de acordo com a aplicação. A Figura 8 ilustra o relacionamento de reações implementadas com a interface Reaction através de um diagrama de classes UML [4]. As reações ilustradas serão apresnetadas em detalhes na seção 4.3.

Em T-Rex, o meta-nível pode ser modificado em tempo de programação ou em tempo de execução. Não há restrição de acesso ao meta-nível. Desta forma, os próprios agentes que fazem parte de uma aplicação podem criar meta-tuplas, e escrevê-las no espaço de meta-nível. O mesmo pode ser feito através da execução de rea-

ções. Sendo assim, novas estratégias de controle podem ser criadas dinamicamente, em tempo de execução.

A visão geral de T-Rex pode ser então descrita através das respostas para as questões de implementação levantadas pelo padrão Reflective Blackboard (Seção 3.9). Estas respostas são apresentadas na Tabela 2.

Como os meta-dados são estruturados?	Os meta-dados são estruturados através de meta-tuplas que associam uma **reação** a uma **operação** efetuada por um agente sobre um **dado**, que na verdade é representado por uma tupla.
Como o MOP é implementado?	O MOP é implementado através de espaços de tuplas reflexivos onde qualquer operação executada no blackboard de nível base é interceptada. Após a interceptação, meta-dados relacionados são procurados em um espaço de meta-nível e caso estes existam, as reações associadas a eles são executadas.
Que componentes terão acesso ao meta-nível?	Não há restrição de acesso ao meta-nível. Desta forma meta-dados podem ser criados em tempo de desenvolvimento ou em tempo de execução. Os próprios agentes que fazem parte da aplicação podem então criar meta-dados.
Que componentes as reações poderão acessar e modificar?	Não há qualquer restrição de acesso às reações. Esta restrição deve ser implementada nas aplicações que utilizam as funcionalidades do framework T-Rex.
Que informações do nível base são reificadas no meta-nível?	As informações relativas ao agente que solicitou uma operação ao espaço de nível base, à operação solicitada e ao dado que foi objeto da solicitação são reificadas no meta-nível e são levadas em consideração na execução das reações.

Tabela 2 Questões de implementação

A dinâmica básica de uma operação efetuada no blackboard de nível base pode ser representada pelo cenário a seguir, e ilustrada na Figura 9 através de um diagrama de seqüência de UML [2].

1. Um agente (Ag1) solicita uma operação ao blackboard de nível base como, por exemplo, a leitura de um dado (representado por uma tupla) passando como parâmetro um padrão de tupla a ser buscada;
2. A tupla buscada é lida no blackboard de nível base através do casamento de padrões. No caso de não existir nenhuma tupla que case com o padrão enviado, o agente solicitante é notificado e o processo interrompido;
3. Caso a tupla buscada no passo 2 exista, o protocolo de meta-objeto (MOP) intercepta a operação e busca no meta-nível, meta-tuplas que possam estar associadas a ela. Para tanto ele efetua uma leitura no espaço de meta-nível buscando meta-tuplas que contenham qualquer reação associada à tripla operação (leitura), agente solicitante (Ag1) e tupla lida;
4. Se a meta-tupla procurada existir, a reação associada a ela é recuperada, os parâmetros interceptados no nível base são reificados no meta-nível

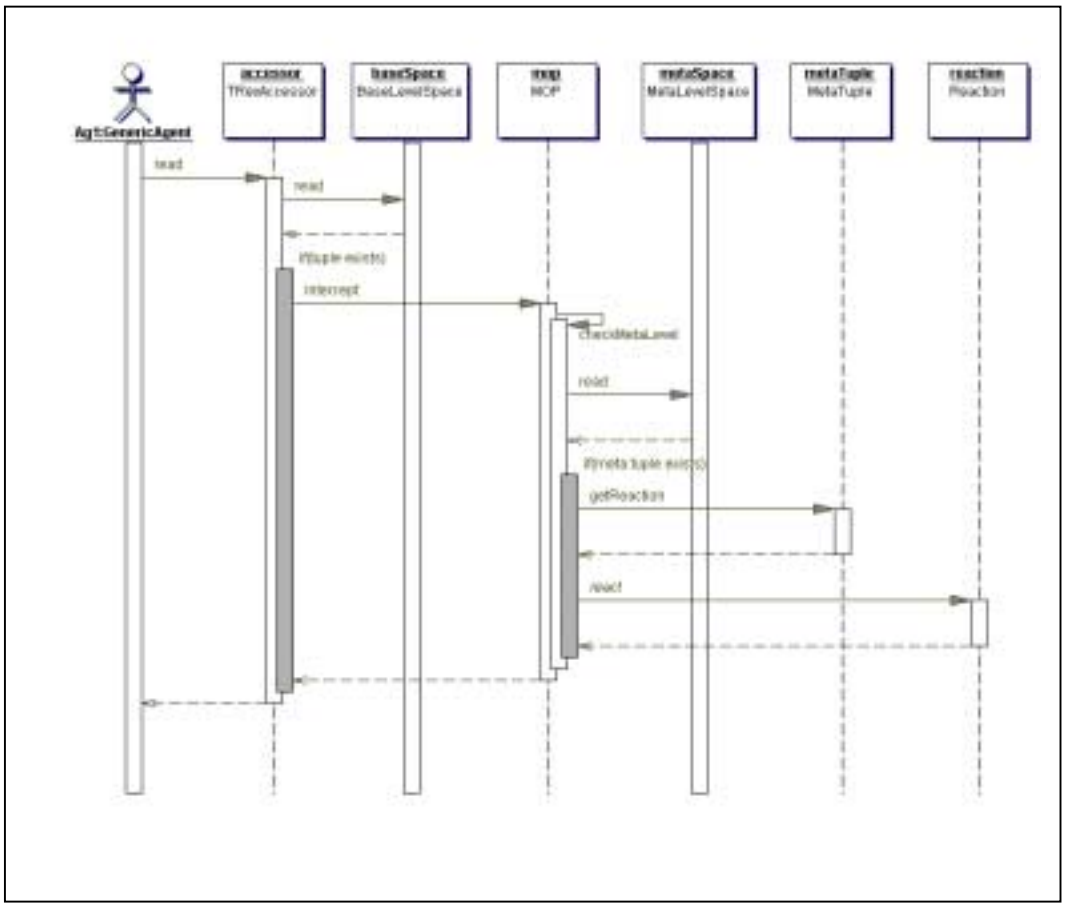

Figura 9 Diagrama de seqüência para a leitura de um dado em T-Rex

e em seguida a reação é executada. O resultado da execução da reação é então retornado ao agente. Caso a meta-tupla procurada não exista no espaço de meta-nível, a própria tupla lida no passo 2 é retornada para o agente.

No caso da solicitação de uma operação de retirada de um dado, o dinâmica é praticamente idêntica à apresentada acima, com a diferença que no passo 2 a tupla buscada é retirada do espaço de nível base ao invés de ser lida. No que se refere à operação de escrita, no passo 3 a mesma tupla é escrita no blackboard de nível base e utilizada na busca por meta-tuplas no meta-nível.

4.1.2 Ambientes de T-Rex

Através da utilização de T-Rex podem ser criados diferentes ambientes em um sistema multi-agente. Cada agente de software situa-se em um ambiente em um determinado instante. Os agentes são capazes de perceber, através de seus sensores, mudanças no ambiente onde estão localizados. Por outro lado eles também podem causar mudanças em seu ambiente através de seus efetuadores. Ambientes distintos

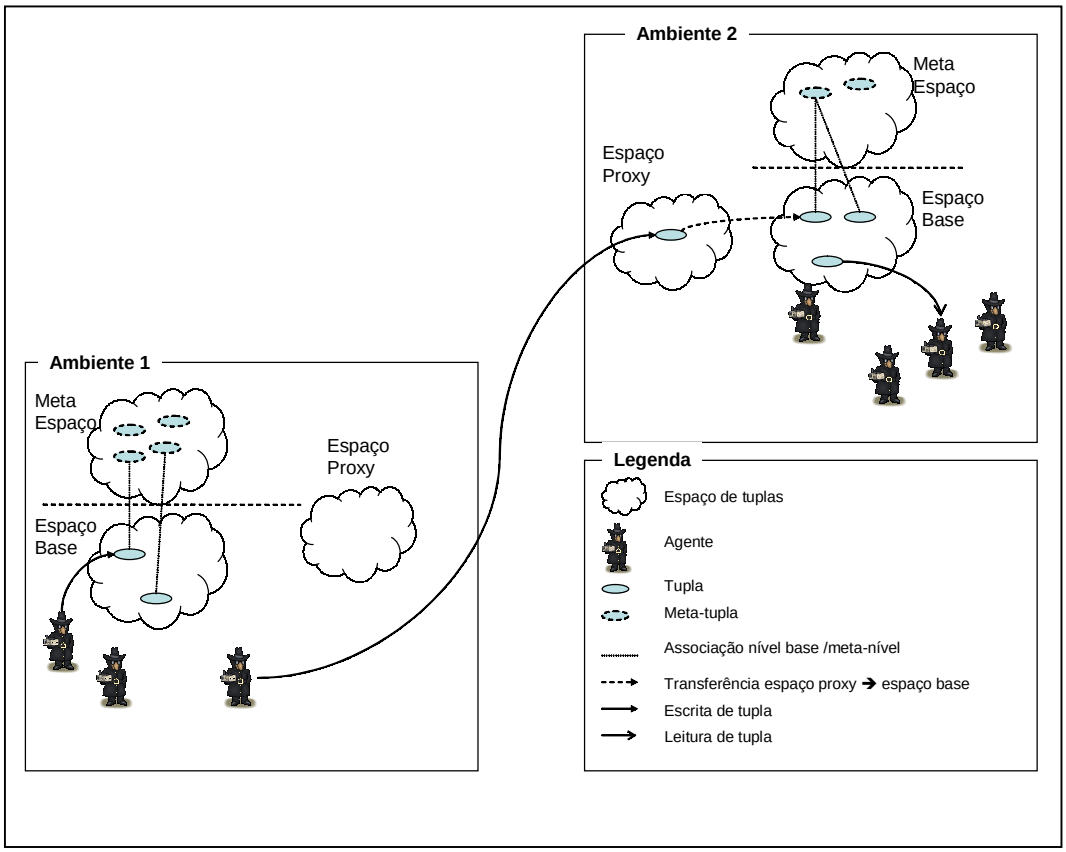

Figura 10 - Os Ambientes de T-Rex

podem estar localizados em uma mesma máquina ou em máquinas diferentes. Mesmo que mais de um ambiente esteja localizado em uma única máquina, todos eles são sempre considerados isolados uns dos outros.

Cada ambiente de T-Rex é composto por três espaços de tuplas distintos. Os dois espaços principais são o espaço base e o meta-espaço, necessários para a implementação do padrão Reflective Blackboard (variante Espaços de Tuplas Reflexivos). Os diferentes agentes são capazes de escrever, ler e excluir tuplas do espaço base. Estas tuplas, por sua vez podem estar associadas a meta-tuplas escritas no meta-espaço. Além destes dois espaços, cada ambiente possui um espaço denominado Espaço Proxy. Este espaço é responsável pelo tratamento de acessos remotos ao ambiente. Desta forma, um agente não é capaz de acessar diretamente o espaço base e o meta-espaço pertencentes a um ambiente no qual ele não está localizado. Todo o acesso é realizado através do Espaço Proxy. Sendo assim, sempre que um agente desejar enviar uma tupla para um ambiente remoto, ele deve enviá-la para o Espaço Proxy deste ambiente, que por sua vez se encarregará da escrita da tupla no espaço base do ambiente de destino. Desta forma, as operações de desvio para o meta-nível e execução de reações são controladas localmente no ambiente de destino do dado. A Figura 10 ilustra os relacionamentos entre os diferentes espaços existentes em um ambiente de T-Rex, assim como seu relacionamento com os agentes de software.

Cada um dos espaços é uma instância distinta de um espaço de tuplas de TSpaces. Para que todos ambientes sejam realmente isolados uns dos outros, é necessário que estes estejam implementados em diferentes servidores TSpaces, ainda que executando na mesma máquina e em portas diferentes.

O acesso ao espaço base e ao meta-espaço é encapsulado por uma única classe denominada TRexAccessor. Esta classe provê todas as operações necessárias para o acesso aos espaços tanto no nível base quanto no meta-nível. São estas as operações interceptadas pelo protocolo de meta-objeto. Cada ambiente provê um ponto central de acesso ao seu TRexAccessor, implementado através do padrão de projeto Singleton [17]. Este padrão tem como objetivo prover um ponto de acesso global para um objeto e garantir que exista apenas uma instância deste objeto.

Os agentes de T-Rex são capazes de se mover de um ambiente para outro. Neste processo, tanto o estado do agente quanto seu código são transferidos. Toda a infra-estrutura de mobilidade é implementada baseada nos próprios espaços de tu-

plas, conforme será apresentado adiante. Sempre que um agente chega em um novo ambiente, ele é capaz de adquirir acesso a ele através do ponto central de aquisição de acesso implementado através do Singleton. A utilização do padrão também garante que todos os agentes localizados em um mesmo ambiente atuem, através de seus sensores e efetuadores, sempre nos mesmos espaços de tuplas.

4.1.3 Agentes de Software em T-Rex

Uma vez que objetivo de T-Rex não é tratar as propriedades intra-agentes o projeto dos agentes de software que utilizam o framework para tratar as propriedades inter-agentes é bastante simples. Esta simplicidade tem como objetivo facilitar a utilização de T-Rex por agentes que utilizem qualquer arquitetura intra-agente [33,21,30].

Desta forma, basta que um agente estenda a classe GenericAgent ou implemente a interface GenericAgentInterface para que ele possa utilizar as funcionalidades disponibilizadas por T-Rex. A disponibilidade de uma interface para a implementação dos agentes possibilita a utilização de T-Rex ainda que seja necessário que o agente estenda outra classe devido à modelagem e implementação de suas propriedades intra-agentes. Esta interface apenas impõe que o agente seja serializável [29] e executável (implementando a interface Runnable de Java). Além disso, ela obriga que o agente possua um identificador padrão AgentID. Este identificador possui, além do nome do agente, uma referência para ambiente T-Rex no qual ele foi criado. Ele é utilizado para endereçar mensagens ao agente bem como para sua identificação durante processos de mobilidade ou quando o agente está persistente. O identificador do agente também é utilizado na solicitação de acesso às operações do ambiente (escrita, leitura e retirada de tuplas) e conseqüentemente no processo de desvio para o meta-nível.

A classe abstrata GenericAgent implementa algumas das operações definidas na interface GenericAgent com o objetivo de simplificar a implementação dos agentes. No entanto, sua utilização pode ser limitada pela necessidade do agente estender alguma outra classe, já que a herança múltipla não é permitida em Java [65].

Devido à simplicidade do projeto dos agentes de T-Rex fica a cargo do programador a implementação de toda a autonomia, adaptação, interatividade e grau de inteligência dos agentes. O programador também é responsável por eventuais preparações dos agentes que venham a ser necessárias antes da solicitação de operações a um ambiente T-Rex, como por exemplo mobilidade e persistência. Por exemplo, o

agente deve estar preparado para retomar seu processamento em um ambiente remoto após sua transferência, ou após ser recuperado de um estado persistente.

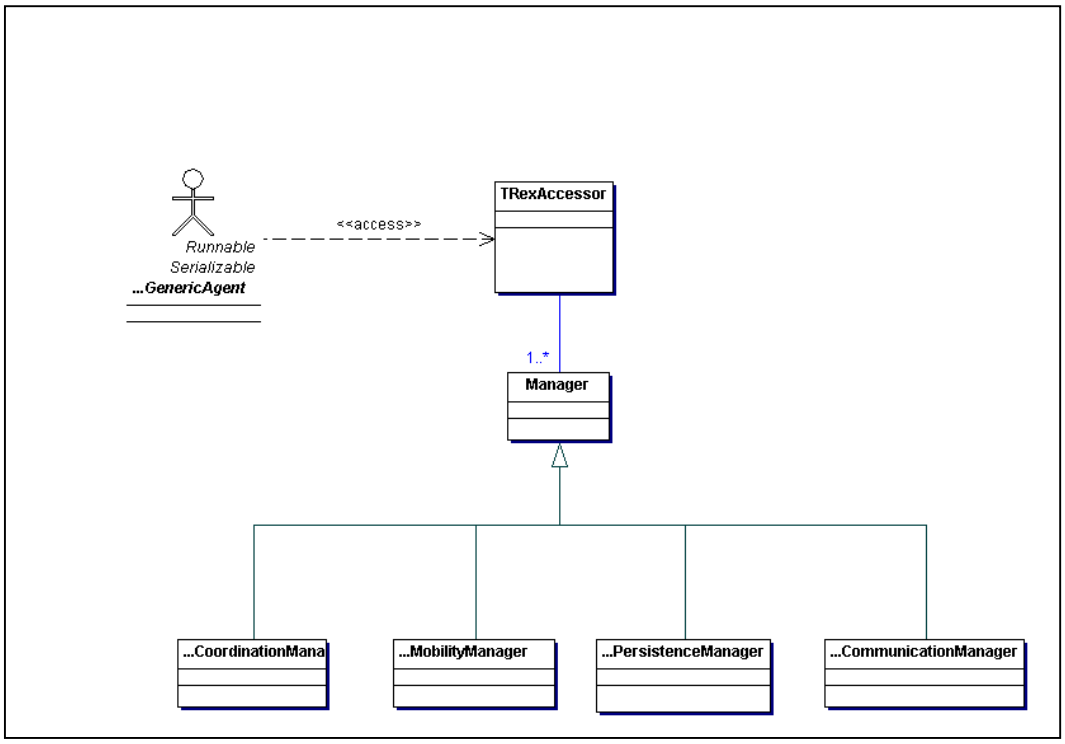

Figura 11 Gerenciadores de propriedades

4.2 Tratamento de Propriedades Inter-Agentes

O framework T-Rex oferece suporte às propriedades inter-agentes de mobilidade, comunicação, coordenação e persistência durante o desenvolvimento e execução de sistemas multi-agentes. Este suporte é baseado em espaços de tuplas reflexivos, o que possibilita um melhor controle e composição destas propriedades [52, 53,54]. De forma a separar as responsabilidades do tratamento das diferentes propriedades foram construídos gerenciadores para cada uma delas. Estes gerenciadores são invocados pelo TRexAccessor quando operações relativas a estas propriedades são solicitadas a ele. A Figura 11 ilustra o projeto destes gerenciadores (*Managers*) através de um diagrama de classes UML. A seguir apresentaremos de maneira mais detalhada a modelagem e implementação do tratamento destas propriedades e eventuais composições existentes.

4.2.1 Mobilidade

A mobilidade dos agentes em T-Rex é implementada baseada nos próprios espaços de tuplas reflexivos. Dois conceitos básicos são utilizados nesta implementação: a possibilidade de acesso remoto a espaços de tuplas e a escrita de objetos seria-

lizáveis nos espaços sob forma de tuplas. Independente da maneira através da qual um agente é modelado [33,21,30], na linguagem Java ele será implementado sob forma de classes e objetos. Mais especificamente em T-Rex, estas classes e objetos devem ser serializáveis. Desta forma, um agente ao solicitar sua transferência de uma ambiente para outro tem seu estado preservado. Em seguida, é transformado em uma tupla que é escrita no espaço de nível base do ambiente de origem e, na seqüência, lida pelo ambiente de destino. Esta tupla é denominada MobileAgentTuple e possui a seguinte estrutura: (“MobileAgent” , AgentID , AgentData , AgentClassName , DestinationTRex), onde “MobileAgent” é a chave que identifica que a tupla contém um agente móvel, AgentID é o identificador do agente, AgentData contém o estado do agente, AgentClassName é o nome da classe que o agente instancia e DestinationTRex é o ambiente de destino do agente.

A escrita da MobileAgentTuple no ambiente de origem, seguindo a dinâmica dos espaços de tuplas reflexivos, é interceptada pelo protocolo de meta-objeto que verifica a existência de meta-tuplas associadas à mobilidade do agente e conseqüentemente executa as eventuais reações associadas. A tupla que contém o agente é escrita em primeiro lugar no seu próprio ambiente porque é ele quem controla, através de estratégias implementadas no meta-nível a partida do agente para outro ambiente.

Após a escrita da tupla que contém o estado preservado do agente (MobileAgentTuple), o ambiente de destino é capaz de solicitar sua leitura e em seguida tornar o agente ativo. Para que o ambiente de destino saiba que existe uma tupla com um agente móvel que deve ser transferida para ele, é enviada uma mensagem de notificação. Esta mensagem, que também é uma tupla, e escrita no espaço Proxy do ambiente de destino, ativando um agente reativo que será o responsável pela leitura e ativação do agente móvel no ambiente de destino. A tupla escrita no espaço Proxy remoto contém informações acerca do identificador e do ambiente de origem do agente móvel. A partir desta tupla o agente reativo MobilityAgent é capaz de executar a leitura da tupla que contém o agente móvel no ambiente de origem. A implementação deste mecanismo baseia-se na infra-estrutura de notificação de eventos presente no IBM TSpaces (seção 3.7).

A mobilidade de um agente depende não somente da mobilidade de seu estado, mas também da mobilidade de seu código [15]. Com a transferência da tupla, apenas o estado do agente é transferido. Para a mobilidade de código, T-Rex se ba-

seia na funcionalidade de carga dinâmica de classes[6] presente na linguagem Java [26]. Através desta funcionalidade, as classes dos agentes, isto é, seu código, devem ficar em um repositório acessível ao ambiente de destino do agente. Desta forma quando o ambiente de destino torna um agente ativo suas classes são recuperadas dinamicamente a partir do repositório comum.

Em T-Rex este repositório, por questão de simplicidade, foi modelado e implementado através de URLs de acesso público localizadas em servidores Web. A utilização dos próprios espaços de tuplas para o armazenamento do código dos agentes também é viável. No entanto, esta última abordagem teria administração mais complexa uma vez que seria necessária alguma ferramenta externa para a escrita do código dos agentes no espaço de tuplas. Por outro lado, para a utilização de URLs como respositório de código basta que as classes sejam armazenadas em um diretório específico do servidor Web.

Esta implementação que leva em consideração a mobilidade de código e de estado dos agentes não é nativa da tecnologia de espaços de tuplas utilizada (IBM TSpaces) e sua implementação faz parte do framework T-Rex. Entretanto esta implementação é transparente para os usuários de T-Rex. Através da utilização da infraestrutura de mobilidade de T-Rex os agentes são recuperados das tuplas e desserializados após a chegada ao espaço de destino. Ao processo de desserialização foram anexadas operações que identificam quais são as classes que contêm o código do agente e as recuperam dinamicamente a partir dos repositórios de classes especificados.

A mobilidade dos agentes em T-Rex pode então ser resumida através dos seguintes passos:

1. O agente toma a decisão de se transferir de uma ambiente para outro;

2. O agente solicita ao TRexAccessor de seu ambiente a sua transferência (Figura 12– Mensagem 1);

3. A responsabilidade da transferência é repassada ao MobilityManager do ambiente (Figura 12 – Mensagem 1.1);

[6] do inglês *dynamic class loading*

4. O MobilityManager serializa o agente sob forma de uma MobileAgentTuple (Figura 12 – Mensagem 1.1.1) e a escreve no espaço de nível base do ambiente (Figura 12 – Mensagem 1.1.2). Após a escrita, o fluxo normal presente no padrão Reflective Blackboard é executado, seguido de eventuais execuções de reações;

5. O MobilityManager cria a GetAgentTuple (Figura 12 – Mensagem 1.1.3), responsável por notificar o ambiente T-Rex de destino que existe um agente que deseja se mover para lá;

6. O MobilityManager escreve a GetAgentTuple no espaço Proxy do ambiente de destino (Figura 12 – Mensagem 1.1.4);

7. Através das funcionalidades de reflexão embutidas no próprio T-Spaces, o agente reativo MobilityAgent é encarregado da transferência e ativação dos agente móvel (Figura 12 – Mensagens 1.1.4.1, 1.1.4.1.1 e 1.1.4.1.2);

8. O MobilityAgent retira a MobileAgentTuple do espaço de nível base do ambiente de origem do agente móvel (Figura 12 – Mensagem 1.1.4.1.2.1);

9. O MobilityAgent solicita a desserialização do agente móvel e as operações relativas à mobilidade de código são automaticamente executadas (Figura 12 – Mensagem 1.1.4.1.2.2);

10. O MobilityAgent reativa o agente móvel no ambiente de destino e este recupera o controle de sua autonomia Figura 12 – Mensagem 1.1.4.1.2.3).

4.2.2 Comunicação

As funcionalidades de comunicação em T-Rex são baseadas na comunicação via Blackboard [3]. Desta forma, as mensagens ou outros dados trocados pelos agentes são escritos no espaço de nível base do ambiente e podem ser lidos em seguida. Este tipo de comunicação preserva a autonomia dos agentes uma vez que eles são capazes de buscar mensagens no blackboard quando desejam.

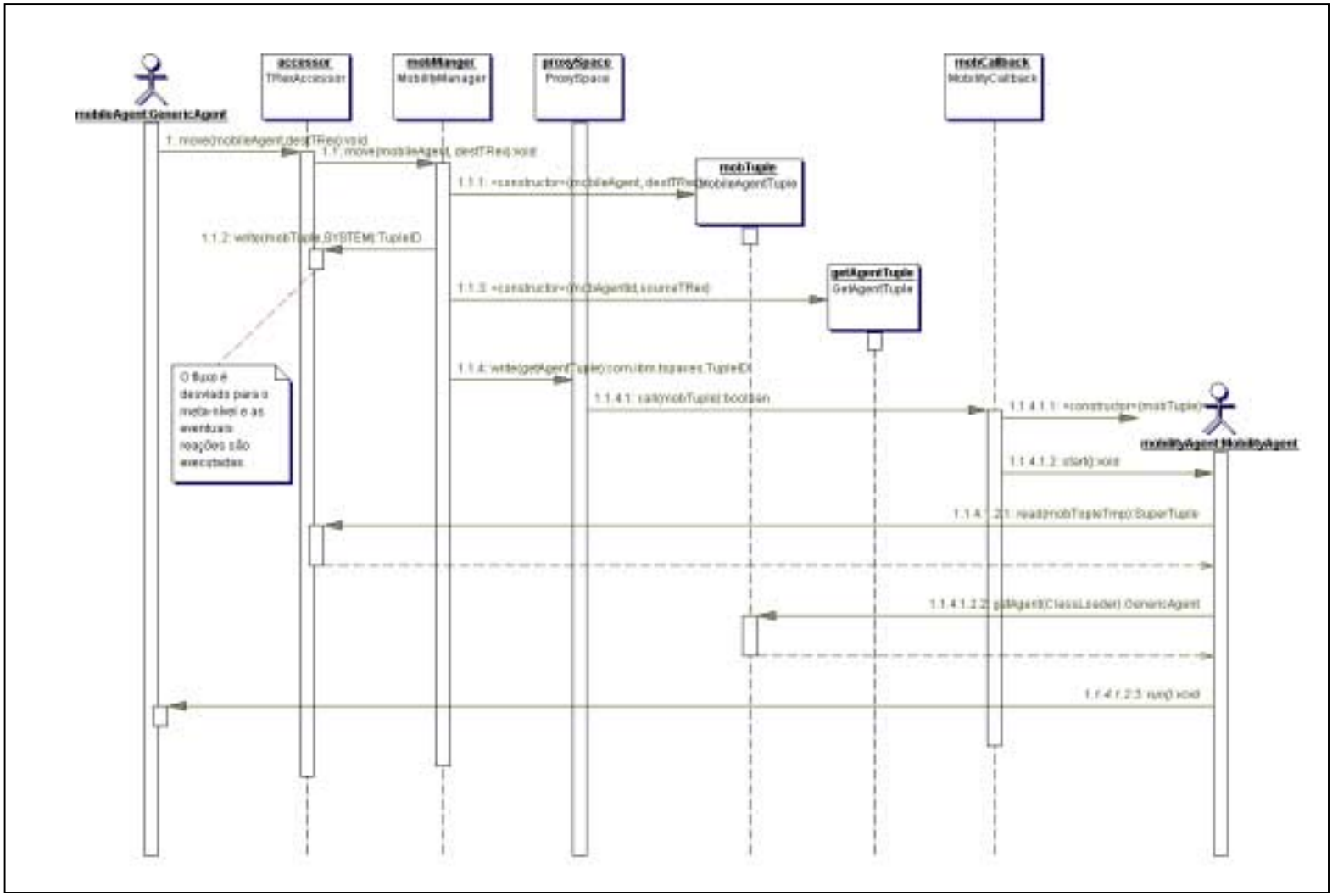

Figura 12 Diagrama de seqüência para a transferência de um agente (mobilidade)

O envio de mensagens para agentes localizados em ambientes remotos é realizado através do acesso ao espaço Proxy do ambiente de destino. Desta forma, sempre que se deseja enviar uma mensagem para um agente que está em um ambiente remoto, deve-se escrever a mensagem no espaço Proxy do ambiente de destino. Assim como no processo de ativação de um agente móvel (Seção 4.2.1) um agente reativo (ativado pelo mecanismo de notificação de eventos presente em TSpaces) é o responsável pela transferência da mensagem do espaço Proxy para os espaço base do ambiente.

Com o objetivo de facilitar a programação deste tipo de envio de mensagens, foi implementado no CommunicationManager, um método que é responsável pelo acesso direto ao espaço Proxy remoto. Desta forma, basta que o agente solicite o envio de uma mensagem através deste método, sendo a criação e o acesso ao espaço Proxy encapsulados pelo gerenciador de comunicação.

A arquitetura reflexiva de T-Rex ainda garante que tanto a escrita quanto a leitura ou retirada de mensagens ou outros dados sejam desviadas para o meta-nível. Desta forma é possível impedir que um agente leia mensagens que não sejam endereçadas para ele mesmo. Este tipo de meta-tupla de proteção de mensagens é criada pelo gerenciador de comunicação na inicialização dos ambientes de T-Rex.

Caso seja necessário para a aplicação podem ser criadas também reações que notifiquem os agentes da existência de mensagens endereçadas a eles. Embora a implementação deste tipo de reação seja simples ela não faz parte da implementação nativa de T-Rex devido à necessidade de utilização do framework independente da forma de implementação das propriedades intra-agentes. Isto porque dependendo da implementação dos agentes, diferentes tratadores de mensagens são utilizados, não sendo possível prever como estes serão chamados.

Outros tipos de reações podem ser utilizados no tratamento da comunicação entre os agentes. Uma das funcionalidades possíveis é a inclusão de reações que sejam responsáveis pela tradução entre mensagens trocadas entre agentes que se comunicam através de linguagens diferentes. Reações também podem ser utilizadas para implementar a composição da propriedade de comunicação com a de mobilidade, executando o roteamento de mensagens para agentes móveis de acordo com o exemplo apresentado no capítulo anterior. A implementação deste exemplo será apresentada com detalhes na seção 4.3.

4.2.3 Persistência

A persistência é outra propriedade inter-agente que é implementada em T-Rex. Esta implementação também está baseada nos espaços de tuplas. Uma vez que estes espaços podem ser configurados para serem persistentes, basta que uma tupla contendo um agente seja escrita em um espaço persistente para que o agente também possua seu estado armazenado.

O tratamento de mensagens endereçadas a agentes persistentes é uma tarefa de certa forma complicada e extremamente importante [42]. Isto porque se a cada mensagem recebida, o agente persistente precisar tornar-se ativo o sistema pode se tornar muito ineficiente [42]. Desta forma, um agente persistente só deve ser ativado quando receber mensagens que são realmente relevantes para ele. Este problema pode ser solucionado através das funcionalidades de reflexão, uma vez que é possível adicionar comportamento à persistência dos agentes. Por exemplo, pode-se criar reações que notifiquem agentes persistentes que mensagens importantes foram enviadas. A detecção do que realmente é uma mensagem importante é realizada através das regras inseridas no meta-nível sob forma de meta-tuplas. No entanto, pela mesma razão apresentada na seção anterior este tipo de regra não faz parte da implementação nativa de T-Rex, já que o framework não pode prever como serão implementados tratadores de mensagens dos agentes que o utilizarão.

Para um agente que utiliza as funcionalidades de T-Rex se tornar persistente basta que ele solicite esta operação ao TRexAccessor. Em seguida o TRexAccessor delegará a responsabilidade da operação ao gerenciador de persistência (PersistenceManager) do ambiente. Este gerenciador será o responsável pela serialização do agente sob forma de uma tupla (PersistentAgentTuple) e sua escrita no espaço de nível base do ambiente. A tupla que armazena o agente possui também informação relativa à identidade do agente (AgentID) de forma a facilitar a associação de meta-tuplas ao agente persistente. A Figura 13 ilustra através de um diagrama de seqüência a operação de tornar um agente persistente.

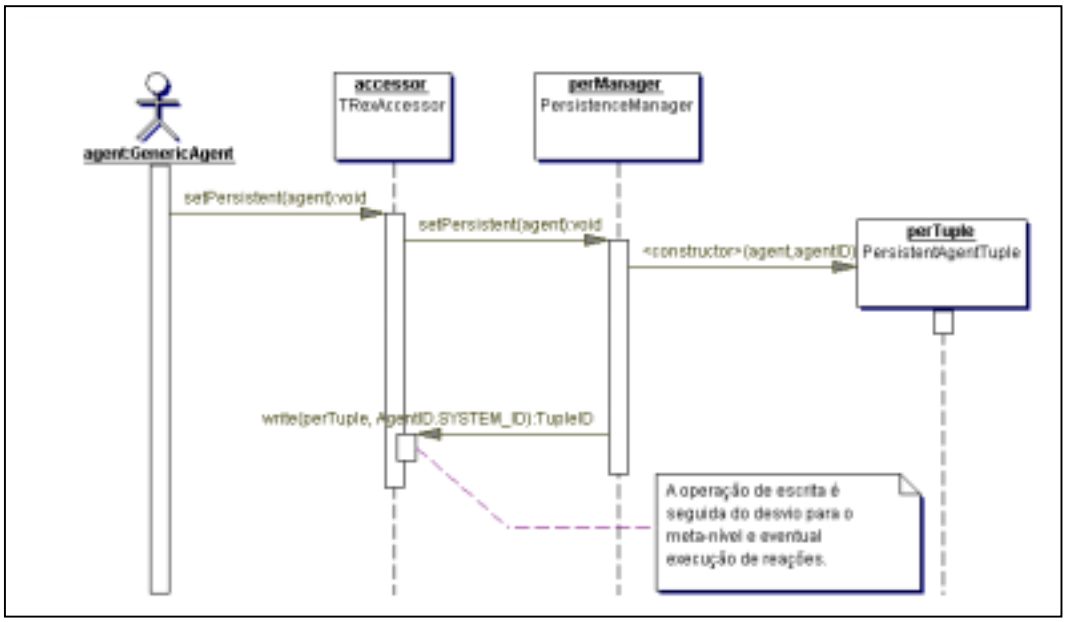

Figura 13 Diagrama de seqüência para a persistência de um agente

Como já indicado anteriormente é de responsabilidade do agente a preparação de seu estado persistente de forma que ele possa retomar seu processamento após a sua reativação. Também é de responsabilidade do agente o tratamento de sua reativação, seja através de meta-tuplas e reações ou de outro mecanismo como, por exemplo, contadores de tempo. A estrutura de T-Rex disponibiliza um método de reativação de agentes, mas não garante que este método será chamado.

4.2.4 Coordenação

Apesar da coordenação geralmente ser uma responsabilidade da aplicação, uma das principais características de T-Rex é oferecer suporte para seu tratamento de forma separada da implementação dos próprios agentes. Esta característica, na verdade, é proveniente da implementação do padrão *Reflective Blackboard* [52,53,54].

Além da própria infra-estrutura reflexiva, T-Rex também oferece suporte para a criação e inserção, através do gerenciador de coordenação, de meta-tuplas e correspondentes reações no meta-nível do ambiente. Os agentes podem assim coordenar suas ações de forma dinâmica, dependendo da aplicação multi-agente implementada. O controle de acesso ao meta-nível não faz parte da implementação nativa de T-Rex, mas esta poderia ser realizada em extensões do framework, por exemplo, através da criação de um meta-meta-nível [8].

De forma a prover algumas funcionalidades avançadas são incorporadas algumas funcionalidades de coordenação ou controle, através de meta-tuplas na imple-

mentação nativa de T-Rex. Uma delas é a implementação do roteamento de mensagens a agentes móveis que ilustra o exemplo de motivação do padrão *Reflective Blackboard* que é apresentado em detalhes na seção 4.3.

4.3 Utilizando T-Rex: Um Exemplo

Para ilustrar em detalhes a utilização de T-Rex no tratamento e composição, através de estratégias de controle, de múltiplas propriedades inter-agentes utilizaremos o exemplo de motivação do padrão Reflective Blackboard (Capítulo 3). Neste exemplo, dois agentes móveis se comunicam através de troca de mensagens e estas mensagens são sempre enviadas para o ambiente natural do agente (a "casa" do agente) que por sua vez se encarrega do direcionamento da mensagem para o ambiente onde o agente se encontra.

Esta implementação em T-Rex se baseia na criação automática de meta-tuplas e reações que inseridas no meta-nível do ambiente representam ponteiros de encaminhamento de mensagens. Estes ponteiros, por sua vez, também são criados através de outros meta-dados que detectam a mudança do agente de um ambiente para outro. Os meta-dados, responsáveis pela detecção da mobilidade são criados e inseridos no espaço de meta-nível pelo gerenciador de mobilidade, durante a inicialização de um ambiente de T-Rex. Estas meta-tuplas então associam uma reação específica (UpMsgPtrRct) à operação de escrita de qualquer MobileAgentTuple no ambiente. O código para esta meta-tupla é apresentado no Fragmento de Código 1.

```
public class UpMsgPtrMetaTuple extends MetaTuple{

    public UpMsgPtrMetaTuple()
        throws TupleSpaceException
    {
        super( new UpdateMsgPtrRct() ,
               //reação responsável pela atualização de ponteiros
               Operation.WRITE ,
               //operação de escrita
               AgentID.class ,
               //coringa para qualquer identificador de agente
               MobileAgentTuple.class );
               //coringa para qualquer MobileAgentTuple
    }
}
```

Fragmento de Código 1 UpMsgPtrMetaTuple

Como a escrita de uma MobileAgentTuple faz parte do processo de mobilidade de um agente (seção 4.2.1) sempre que um agente se mover de um ambiente para

outro, a execução será desviada para o meta-nível e a reação UpdateMsgPtrRct será executada. Esta reação identifica o agente que está se movendo, bem como o seu ambiente original. Após a identificação, é inserido um ponteiro de encaminhamento no meta-nível do ambiente natural do agente que está se movendo. Caso já exista algum outro ponteiro relativo a este mesmo agente no meta-nível de seu ambiente natural, este será removido e substituído pelo novo ponteiro. Cabe lembrar que através desta estratégia, não são criadas cadeias de ponteiros, mas apenas um ponteiro localizado no ambiente natural do agente seguindo a idéia proposta pelo protocolo Mobile IP [ref]. Na verdade, este processo de atualização de ponteiros pode ser comparado ao processo de "Binding Update" presente no Mobile IP. O Fragmento de Código 2 apresenta a implementação do método react desta reação.

```
public SuperTuple react( SuperTuple t,Operation op,AgentID id )
    throws TRexException
{
    MobileAgentTuple mobTuple = ( MobileAgentTuple ) t;
    //recuperando o identificador do agente
    AgentID mobAgentId = mobTuple.getID();
    //retirando ponteiros antigos
    MessagePointer oldMessagePointerRule =
        new MessagePointer( mobAgentId );
    MetaLevelSpace homeMetaSpace =
        new MetaLevelSpace( mobAgentId.getHome() );
    SuperTuple oldPointer =
        homeMetaSpace.take( oldMessagePointerRule );
    //criando novo ponteiro
    MessagePointer pointer =
        new MessagePointer( mobAgentId ,
                            mobTuple.getDestination() );
    //escrevendo novo ponteiro
    homeMetaSpace.write( pointer );
    return t;
}
```

Fragmento de Código 2 Método react da reação UpdateMsgPtrRct

A partir do processo de atualização automática de ponteiros é possível ter, sempre que o agente se moveu para outro ambiente, um ponteiro de encaminhamento de mensagens. Este ponteiro associa uma reação (MessagePointerReaction) a toda operação de escrita de mensagens endereçadas ao agente móvel ao qual o ponteiro se refere, não levando em consideração o agente que escreveu a mensagem. O Fragmento de Código 3, apresenta o código comentado da meta-tupla que representa o ponteiro de encaminhamento de mensagens.

```
public class MessagePointer extends MetaTuple
{

    public MessagePointer( AgentID id ,
                           SpaceInfo destSpaceInfo )
    throws TupleSpaceException
    {
        super( new MessagePointerReaction( destSpaceInfo ) ,
               //reação responsável pelo encaminhamento das
               //mensagens para o espaço de destino
               Operation.WRITE ,
               //operação de escrita
               AgentID.class ,
               //coringa para o agente que escreveu a
               //mensagem
               new TRexMessage( id ) );
               //qualquer mensagem endereçada para o agente
               //ao qual o ponteiro se refere
    }
}
```

Fragmento de Código 3 Message Pointer

A execução da reação MessagePointerReaction transfere a mensagem para o ambiente onde o agente móvel realmente se encontra. A própria reação armazena, desde o momento da sua criação, informações acerca do ambiente onde o agente móvel ao qual ela se refere está localizado. Durante sua execução, ela então remove a mensagem do ambiente natural do agente e a escreve no ambiente onde ele realmente está localizado. A implementação do método react desta reação é apresentada no Fragmento de Código 4.

```
public SuperTuple react(SuperTuple t, Operation op, AgentID id)
    throws TRexException
{
  //adquirindo acesso ao espaço proxy do ambiente onde o agente
  //está localizado
  //destSpaceInfo é a informação relativa à localização do
  //agente recebida no momento da criação da reação
  ProxySpace pspace = new ProxySpace( destSpaceInfo );
  //retirando a mensagem do ambiente natural
  TRexAccessor accessor = SingletonTRexAccessor.getInstance();
  BaseLevelSpace baseSpace = accessor.getBaseSpace();
  baseSpace.take( t );
  //escrevendo a mensagem no ambiente onde o agente está
  tspace.write( t );

  return t;
}
```

Fragmento de Código 4 Método react da reação MessagePointerReaction

Este exemplo mostra como estratégias de controle podem ser implementadas de maneira simples e de forma independente da implementação dos agentes em si. Através de implementação destas estratégias no meta-nível do sistema, sua manutenção se torna bastante simples uma vez que basta, por exemplo, alterar a reação associada ao ponteiro de encaminhamento de mensagens para modificar toda a estratégia de controle da comunicação em conjunto com a mobilidade. Além disso, caso não se deseje utilizar nenhuma estratégia de encaminhamento de mensagens basta remover (na verdade não adicionar) a tupla de atualização de ponteiros do meta-nível de cada ambiente. Este tipo de configuração pode ser realizado dinamicamente pelo administrador do sistema multi-agente ou até pelos próprios agentes.

Este exemplo também mostra que a reutilização de estratégias de controle também é simplificada pelo uso da abordagem aplicada. Independente da aplicação ou da forma de implementação dos agentes as estratégias de encaminhamento de mensagens, implementadas em um nível separado da lógica dos agentes e das próprias mensagens, podem ser utilizadas. A estratégia apresentada neste exemplo é incorporada na implementação nativa de T-Rex, podendo ser utilizada por todas as aplicações multi-agentes que a utilizem como base.

Cabe lembrar que este exemplo visa apenas mostrar a utilidade da implementação de estratégias de controle em T-Rex. Não faz parte de seu objetivo garantir que todas as mensagens serão entregues aos agentes, uma vez que podem ocorrer situações específicas onde mensagens são perdidas. Estas situações se referem principalmente à existência de ponteiros que ainda não foram atualizados quando uma mensagem é enviada para um agente. Neste caso, a aplicação deveria tratar estas mensagens que não atingem seu destino.

4.4 Uma Aplicação de T-Rex: Marketplace

Para validar a utilização de T-Rex com múltiplas estratégias de controle foi desenvolvida uma aplicação de Marketplace [1,2,27,55] baseada em uma abstração da vida real. Nesta aplicação, diferentes agentes compradores são responsáveis pela compra de itens em um mercado central. Cada agente comprador representa uma empresa e responde a requisições de usuários desta empresa. Por exemplo, um agente comprador pode responder a uma requisição de compra de toner para impressora efetuada por um agente requisitante do departamento administrativo de sua empresa.

Após o recebimento da requisição, o agente vai ao marketplace com o objetivo de efetuar a compra do item. Se o item estiver disponível no marketplace, o agente verifica se seu preço é satisfatório e em caso positivo realiza a compra. Após a compra do item, o agente o leva consigo para armazená-lo na sua empresa. Na verdade, a compra de um item representa o envio de uma ordem de compra para o fornecedor do item enquanto o armazenamento do item é representado pelo armazenamento de informações acerca do item, como, por exemplo, data de entrega e fornecedor escolhido, nos sistemas de gestão internos da empresa do agente.

Quando os agentes compradores não têm nenhuma requisição para responder, eles ficam inativos e vão dormir. No entanto, eles devem ser capazes de responder de imediato qualquer requisição enviada para eles. Desta forma o envio de uma requisição deve acordar um agente comprador que estava até então dormindo.

Os itens vendidos ficam expostos assim como seus preços no Marketplace. Não existe nenhum tipo de agente vendedor, capaz de negociar os preços com os agentes. Sendo assim os preços dos itens expostos são fixos. O Marketplace possui também regras de negociação que são específicas dele. Por exemplo, podem existir regras que limitem a visibilidade de propostas para vários agentes ao mesmo tempo escondendo-as após um determinado número de consultas. Outras regras podem impedir a compra de itens por agentes sem crédito no mercado. Estas regras podem sofrer alterações constantes dependendo do cenário político-econômico em questão. No entanto é importante que as regras não causem alterações na maneira dos agentes trabalharem. O Marketplace também possui interesse em armazenar relatórios gerenciais e informativos a respeito das transações efetuadas nele, para posterior uso de seus clientes.

4.4.1 Os Agentes da Aplicação

Da maneira que a aplicação de Marketplace foi modelada e implementada existem dois tipos diferentes de agentes. Os agentes requisitantes e os agentes compradores. A seguir estes agentes serão apresentados.

4.4.1.1 Agentes Requisitantes

Os agentes requisitantes representam os empregados de uma empresa que necessitam de diferentes materiais para realizar seu trabalho. Desta forma quando existe a necessidade de algum item ele envia uma mensagem, contendo a requisição do item desejado, para o agente comprador, que por sua vez se encarregará da compra.

O agente comprador então passa a toda sua “vida” requisitando itens para o seu trabalho. Esta requisição é baseada em uma função de necessidade intrínseca ao agente. Na aplicação implementada esta função é bastante simples, uma vez que não é seu objetivo desenvolver agentes complexos e com inteligência. A função basicamente escolhe de maneira aleatória um dos itens que faz parte da lista de itens desejados pelo agente requisitante. Desta forma a autonomia do agente requisitante se resume a enviar a cada intervalo de tempo definido, uma requisição de um item aleatório para o agente comprador.

O agente requisitante não sabe a princípio a localização do agente comprador que pode tanto estar na empresa, quanto no Marketplace. No entanto para que haja um melhor funcionamento do processo de compras é importante que mensagens enviadas para o agente comprador quando ele está no Marketplace sejam enviadas diretamente para lá. Desta forma o agente comprador pode economizar tempo e recursos em transferências do Marketplace para a empresa e vice-versa.

4.4.1.2 Agentes Compradores

Os agentes compradores são então os responsáveis pela compra no Marketplace central dos itens requisitados. Além da compra ele se responsabiliza pelo armazenamento do item na sua empresa. A vida de um agente comprador na aplicação implementada pode ser resumida em três estados distintos: OCIOSO, Comprando e Armazenando. A transição entre estes estados é ilustrada através de um diagrama de atividades UML na Figura .

Inicialmente o agente está OCIOSO, ou seja, sem requisições de compra para responder. Neste estado ele verifica se existem mensagens para ele contendo requisições enviadas pelos agentes requisitantes. Caso não existam requisições ele continua ocioso e tornando-se persistente. Caso existam requisições a serem respondidas, seu estado é alterado para COMPRANDO. Quando o agente comprador está persistente ele deve ser capaz de se tornar ativo sempre que mensagens contendo requisições forem enviadas a ele. Caso outras mensagens sejam enviadas ele continuará persistente

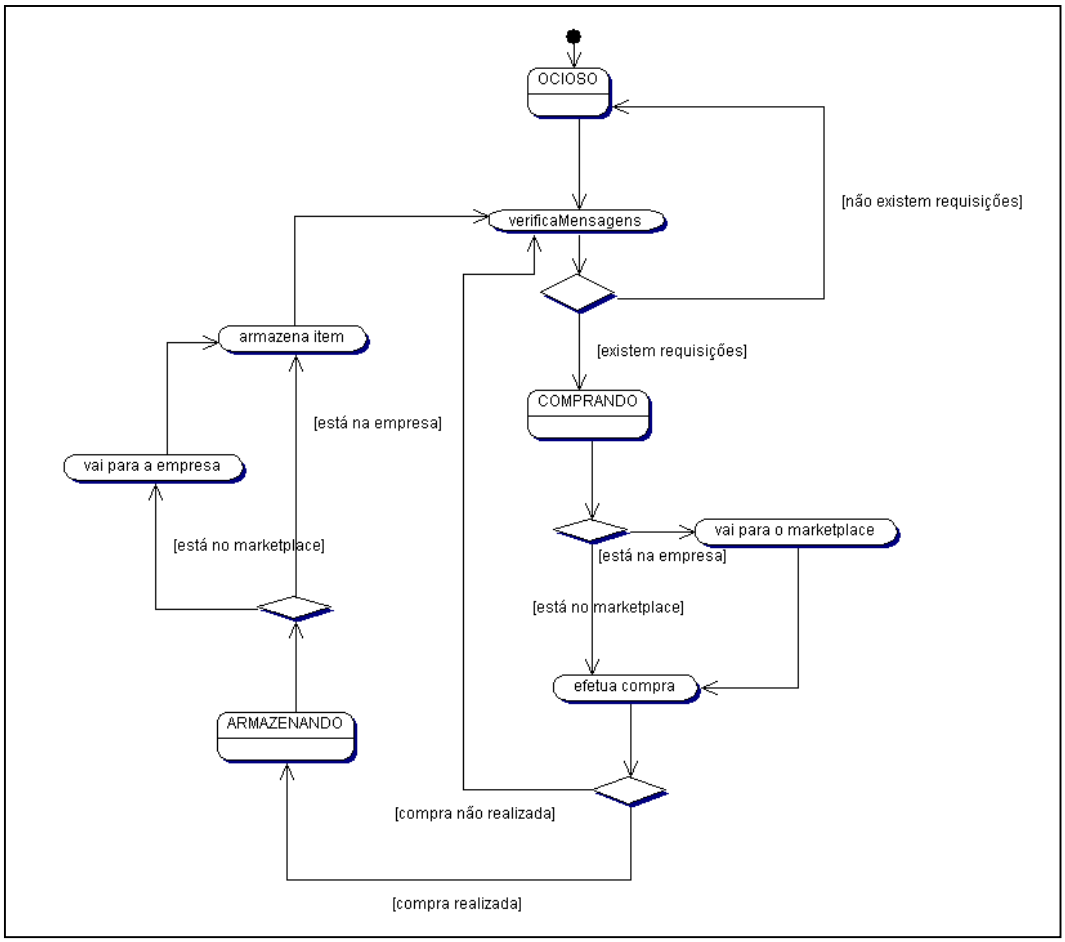

Figura 14 Diagrama de atividades para o agente comprador

Quando o agente está no estado COMPRANDO, ele deve ir para o Marketplace realizar a compra dos itens requisitados. O agente então verifica se ele já está no Marketplace e em caso negativo, se transfere para lá. Uma vez no Marketplace, o agente pode tentar efetuar a compra do item requisitado. Para tanto ele verifica a disponibilidade do item e seu preço e decide se fará a compra ou não baseado em uma função de utilidade. Mais uma vez, como não é o objetivo desta aplicação a elaboração agentes inteligentes, esta função de utilidade simplesmente verifica se o agente comprador possui dinheiro suficiente para a compra do item. Este dinheiro é, na verdade, um atributo do agente, inicializado aleatoriamente no instante de sua criação. Caso o item requisitado esteja disponível, e o agente possua dinheiro suficiente para comprá-lo, a compra é realizada. Neste caso o estado do agente é alterado para ARMAZENANDO. Caso contrário ele verifica se existem novas mensagens com requisições enviadas para ele, se existirem ele continua COMPRANDO. Caso não exista mais nenhuma requisição ele volta para o estado OCIOSO.

Quando o agente está no estado ARMAZENANDO, ele deve retornar a sua empresa para realizar o armazenamento do item comprado. Desta forma ele verifica

se já está na sua empresa para que possa armazenar o item. Caso ele ainda esteja no Marketplace, ele se transfere para a sua empresa. Uma vez na empresa, ele pode então armazenar o item comprado. Após o armazenamento o agente novamente verifica se existem novas mensagens com requisições a serem respondidas. Caso não exista nenhuma ele se torna OCIOSO. Caso contrário seu estado é alterado para COMPRANDO.

4.4.2 Implementando a Aplicação com T-Rex

Através da utilização de T-Rex o Marketplace é implementado como um ambiente de T-Rex enquanto as diferentes empresas são outros ambientes. Cada ambiente é localizado em um servidor distinto, ou seja, os ambientes das empresas ficam dentro das próprias empresas enquanto o marketplace fica em um servidor de acesso comum a todos os agentes. Desta forma, quando um agente comprador vai às compras ele está realmente se movendo da sua empresa para o Marketplace. Por outro lado, quando um agente efetua uma compra e deseja armazená-lo ele se move do Marketplace para sua empresa.

Os itens vendidos ficam expostos no Marketplace em conjunto com seus preços. Estes itens são representados por tuplas escritas no espaço de nível base do ambiente T-Rex que representa o Marketplace. Estas tuplas são instâncias da classe ProposalTuple e armazenam além do item, o seu preço e o nome do fornecedor do item. Desta forma, quando os agentes compradores buscam por um item no Marketplace eles estão realizando a leitura de uma tupla de proposta para o item procurado e com qualquer preço. Caso exista alguma proposta de venda para o item procurado, após sua análise, o agente pode decidir comprar o item. A compra do item é realizada através da retirada da tupla que contém a proposta para o item do ambiente.

Através da estrutura reflexiva de T-Rex é possível associar reações às tuplas que contêm as propostas. Desta forma podem ser programadas ações que serão tomadas após a compra de um item por um agente. Na aplicação implementada as ações tomadas estão relacionadas ao envio de uma ordem de compra para o fornecedor do item e à geração de relatórios de atividades do Marketplace. Isto é realizado inserindo-se no meta-nível do ambiente do Marketplace uma meta-tupla que esteja associada a qualquer retirada de tuplas de propostas por qualquer agente. A reação associada à meta-tupla, por sua vez, através da reificação dos dados do agente comprador, e da tupla retirada é capaz de enviar em nome do comprador uma ordem de

compra ao fornecedor relativo à proposta. A mesma reação é responsável pela inclusão dos dados da ordem de compra nos relatórios de atividades do Marketplace. Como ambas ações são então implementadas no meta-nível do sistema através de meta-tuplas e podem ser modificadas sem interferir no comportamento normal dos agentes. Este tipo de modificação pode ser executado em tempo de execução do sistema, simplesmente através da troca de uma meta-tupla por outra tornando desnecessário que o Marketplace saia do ar durante este tipo de manutenção. O código fonte para estas meta-tuplas e reações pode ser observado no Apêndice I.

As meta-tuplas também podem ser utilizadas para associar regras específicas do Marketplace. Na aplicação desenvolvida foi criado um tipo de meta-tupla que impede a compra de uma quantidade excessiva de um item por um mesmo agente. Esta regra está relacionada ao racionamento de consumo de um certo item. Esta meta tupla associa uma reação à retirada (ou compra) efetuada por qualquer agente de uma proposta específica, como por exemplo, a proposta de venda de botijão de gás. Desta forma pode ser configurado o racionamento no consumo de um bem determinado. A reação associada a esta meta-tupla armazena os agentes que já compraram botijão de gás outras vezes e os impedem de comprar o item mais uma vez. Este impedimento é na verdade representado pela não retirada do ambiente da tupla que contem a proposta e a sinalização de que tal tupla não está disponível para o agente. Mais uma vez, como estas regras estão implementadas no meta-nível elas podem ser alteradas ou “desligadas”, de maneira simples e em tempo de execução. O código fonte relativo à implementação desta regra do Marketplace também pode ser encontrado no Apêndice I.

Os agentes compradores presentes na aplicação devem se mover de suas empresas para o Marketplace e vice-versa. Estes agentes então utilizam as funcionalidades de mobilidade disponíveis na infra-estrutura de T-Rex para realizar sua transferência entre os diferentes ambientes, conforme apresentado na seção 4.2.1.

Os agentes requisitantes enviam mensagens para os agentes compradores, estejam eles na própria empresa ou no Marketplace. Desta forma as mensagens contendo as requisições devem ser enviadas para o ambiente T-Rex onde o agente comprador realmente está. Por outro lado se as requisições forem enviadas para o Marketplace elas somente poderão ser acessadas pelo agente comprador que representa a empresa do requisitante. Desta forma agentes compradores concorrentes não poderão ter acesso a informações relativas às compras de outras empresas, que na maioria das

vezes são confidenciais. Meta-tuplas de controle distintas são responsáveis pelo encaminhamento das mensagens aos agentes (Seção 4.3) e pela segurança destas mensagens. (Seção 4.2.2). No que se refere ao código fonte destas regras ele pode ser encontrado na seção 4.3 para as regras de encaminhamento de mensagens e no Apêndice I para as meta-tuplas relativas à segurança das mensagens.

Meta-tuplas também são utilizadas para reativar agentes que estão inativos e persistentes. Estas meta-tuplas são criadas e inseridas no meta-nível pelos agentes antes de terem seu estado preservado e armazenado de forma persistente. Elas associam qualquer mensagem enviada para o agente à uma reação responsável pela sua reativação, conforme descrito na seção 4.2.3. O código fonte relativo à implementação destas meta-tuplas na aplicação desenvolvida pode ser encontrado no Apêndice I.

4.5 Trabalhos Relacionados

Existem algumas outras arquiteturas e pacotes que podem ser utilizados para tratar as propriedades inter-agentes. A seguir apresentaremos uma comparação destas arquiteturas com a implementação de T-Rex.

4.5.1 MARS

MARS (Mobile Active Reactive Spaces) [9] é uma implementação de espaços de tuplas reflexivos baseada na infra-estrutura do pacote JavaSpaces [14]. MARS define um framework para a coordenação de agentes móveis através de espaços de tuplas programáveis ou reativos. O modelo reflexivo implementado em MARS é bastante semelhante ao de T-Rex sendo as meta-tuplas, o MOP e a reificação de informações no meta-nível estruturados de maneira quase idêntica.

Por outro lado, o desenvolvimento e modelagem da MARS não se baseia nos conceitos teóricos de Reflexão Computacional nem em padrões arquiteturais. Na verdade, acredita-se que o padrão Reflective Blackboard poderia ser aplicado a MARS para que sua arquitetura fosse mais bem estruturada. A utilização dos conceitos de reflexão e padrões arquiteturais no desenvolvimento provê a T-Rex um embasamento teórico e vocabulário padrão inexistente em MARS, o que facilita a sua compreensão por engenheiros de software que desejam utilizá-lo.

MARS pode ser utilizado apenas na coordenação das atividades dos agentes e necessita da utilização de outros pacotes, como por exemplo, Aglets [34], para implementar a mobilidade e outras propriedades inter-agentes. Desta forma os agentes

devem informar de forma explícita para a estrutura de MARS, ações como a mudança de um ambiente para outro para que estas possam ser coordenadas. Por outro lado, uma vez que T-Rex se propõe a tratar de forma unificada as diferentes propriedades inter-agentes, tal tipo de notificação explícita torna-se desnecessária. Isto porque as ações de tornar um agente persistente, enviar mensagens para outros agentes, de transferir um agente de um ambiente para outro passam de forma automática pelos espaços de tuplas dos ambientes, uma vez que estes são a base da implementação destas operações.

Outro ponto facilitado pela utilização de T-Rex é que sua utilização é mais simples que a de MARS. Enquanto desenvolvedores de aplicações baseadas em T-Rex necessitam apenas aprender e familiarizar-se com a utilização do framework, os desenvolvedores de aplicações baseadas em MARS necessitam conhecer além do próprio MARS outro pacote de implementação de propriedades inter-agentes como por exemplo Aglets [34].

4.5.2 TSpaces

O pacote TSpaces [35] também pode ser utilizado sozinho como forma de implementação de algumas propriedades inter-agentes, utilizando seu mecanismo de reflexão nativo como forma de programar as ações a serem executadas pelo meta-nível. No entanto o mecanismo nativo de reflexão implementado por TSpaces não leva em consideração o agente que executou a operação que causou uma reação, sendo então mais pobre que o disponibilizado por T-Rex. O desenvolvimento de T-Rex poderia ter se baseado neste mecanismo para desenvolver seu protocolo de meta-objeto, mas preferiu-se desenvolver um outro diferente (baseado em espaços distintos no nível base e no meta-nível) para que se pudesse ter mais controle sobre a implementação.

TSpaces também não provê nenhum suporte nativo à mobilidade dos agentes enquanto T-Rex possui um mecanismo de mobilidade desenvolvido sobre o próprio TSpaces. Embora TSpaces tenha se mostrado como uma ferramenta útil no desenvolvimento de frameworks para troca de mensagens entre agentes [47], não há nada que se possa fazer em TSpaces que não possa ser desenvolvido em T-Rex, uma vez que T-Rex estende as funcionalidades de TSpaces.

TSpaces assim com o MARS não utiliza qualquer abordagem baseada em padrões na sua arquitetura, desenvolvimento ou documentação. De qualquer forma a

utilização do padrão Reflective Blackboard também poderia ser aplicada no pacote TSpaces com objetivo de documentação ou refatoramento[7] [13].

4.5.3 Aglets

O framework Aglets [34] provê uma série de funcionalidades para modelagem e implementação de agentes móveis. Através da utilização de Aglets é possível desenvolver agentes móveis com uma certa facilidade, uma vez que toda a infra-estrutura de transferência de agentes de um servidor para outro está disponível. Aglets também disponibiliza mecanismos de trocas de mensagens entre agentes que estão situados em servidores diferentes, de forma transparente para os agentes.

A programação do comportamento dos agentes em Aglets é orientada a eventos. Desta forma são construídos diferentes tratadores para cada um dos eventos que podem ocorrer no sistema multi-agente. Como exemplos de eventos pode-se citar a transferência de um agente de um ambiente para outro ou o recebimento de mensagens pelos agentes. Este tipo de implementação, apesar de simples restringe o tipo de modelagem das propriedades intra-agentes dos agentes de um sistema multi-agente baseado em Aglets, uma vez que elas devem ser sempre modeladas a partir de uma abordagem orientada a eventos. Por outro lado a utilização de T-Rex com outras formas de modelagem e implementação intra-agentes, até mesmo Aglets, é simplificada. Isto pode ser dito porque T-Rex apenas impõe que os agentes implementem uma interface específica e que possuam um identificador padrão. Desta forma a responsabilidade de tratamento das propriedades inter-agentes e intra-agentes é dividida entre T-Rex e a metodologia de implementação dos agentes em si respectivamente [50].

Aglets também não possui nenhuma abordagem para o tratamento das estratégias de controle do sistema multi-agente. Desta forma sua utilização depende da criação de ambientes responsáveis pela coordenação ou de agentes coordenadores, sendo que esta prática não se encaixa bem na maioria dos sistemas multi-agentes [9].

[7] Do inglês *refactoring*