UNIVERSIDADE DO VALE DO RIO DOS SINOS

CIÊNCIAS EXATAS E TECNOLÓGICAS

PROGRAMA INTERDISCIPLINAR DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO APLICADA

Holo*Go* : Um Modelo de Mobilidade de Código Orientado ao Holoparadigma

por
Gustavo Lermen

Prof. Dr. Jorge Luis Victória Barbosa
Orientador

Dissertação submetida à avaliação
como requisito parcial para a obtenção do grau de
Mestre em Computação Aplicada

São Leopoldo, janeiro de 2007

Ficha catalográfica elaborada pela Biblioteca da
Universidade do Vale do Rio dos Sinos

L616h Lermen, Gustavo
Holo*Go*: um modelo de mobilidade de código orientado ao holoparadigma / por Gustavo Lermen. -- 2007.
91 f. : il. ; 30cm.

Dissertação (mestrado) -- Universidade do Vale do Rio dos Sinos,
Programa de Pós-Graduação em Computação Aplicada, 2007.

"Orientação: Prof. Dr. Jorge Luis Victória Barbosa, Ciências Exatas
e Tecnológicas".

1. Computação móvel. 2. Mobilidade forte de código. 3. Linguagem de programação. 4. Holoparadigma. I. Título.

CDU 004.75.057.5

Catalogação na Publicação:
Bibliotecária Eliete Mari Doncato Brasil - CRB 10/1184

“Todas as coisas são difíceis antes de se tornarem fáceis”

John Norley

Agradecimentos

Ao meu orientador Prof. Dr. Jorge Luís Victória Barbosa pela motivação e empenho durante o curso.

Aos meus pais, Aloir Lermen e Lisete Lermen, pela paciência, apoio e dedicação que foram imprescindíveis durante o andamento do curso.

A minha namorada Luciana Ribeiro Freitas, pelo amor, amizade e paciência nos momentos em que estive ausente.

Aos meus amigos e colegas do Mobilab: Cícero Raupp Rolim (Baboo), Darci Levis, Sólon Rabello, Rodrigo Hahn, Renato Costa e Fabiane Cristine Dillemburg.

A todos meus colegas de curso e a todos, que de alguma forma, contribuíram para a realização deste trabalho.

A Hewlet-Packard Computdores (HP) pela bolsa de mestrado e incentivo financeiro.

Resumo

A rápida popularização de dispositivos computacionais cada vez mais poderosos está trazendo a computação móvel para um grande número de pessoas. Na medida em que os dispositivos ganham mais poder computacional, diferentes aplicações podem ser desenvolvidas de modo a atender as necessidades de um número cada vez maior de usuários. Esta evolução, entretanto, não é livre de desafios. Com o aumento do número de usuários, a demanda por customização torna-se uma questão chave. Entre as soluções que oferecem a customização encontra-se a mobilidade de código. Neste sentido esta dissertação apresenta um modelo de mobilidade forte de código orientado ao Holoparadigma. Até então somente a mobilidade lógica era suportada, estando à mobilidade de código apenas na especificação. Este modelo é chamado Holo*Go* e foi desenvolvido tendo como base a HoloVM, uma máquina virtual com suporte a *blackboards* e programação concorrente. A validação deste modelo ocorreu através da implementação de um protótipo que foi utilizado no desenvolvimento de aplicações que utilizam à mobilidade de código. Neste contexto a principal contribuição deste trabalho consiste da adição da funcionalidade de mobilidade forte de código ao ambiente do Holoparadigma.

Palavras chave: Mobilidade forte de código, linguagens de programação, máquinas virtuais, computação móvel.

TITLE: HOLO*GO* : “A STRONG CODE MOBILITY MODEL FOCUSED ON THE HOLOPARADIGM”

Abstract

The rapid popularization of increasingly powerful computational devices is setting the mobile computing into daily life of a great number of people. As devices evolve along with its computational power, many applications can be developed in order to accomplish the different needs of a growing number of users. This evolution, however, it is not free of challenges. As the number of users increase, the need of customization becomes a major concern. Among the solutions that offer customization to software applications is code mobility. In this way, this dissertation presents a strong code mobility model focused in the Holoparadigm. Until now only logical mobility was provided, being the code mobility only in the specification. This model is called Holo*Go* and was developed on top of HoloVM, a virtual machine with blackboards and concurrent programming support. Holo*Go* was validated through the implementation of a prototype used in the development of applications that use code mobility. In this context, the main contribution of this work comprises the addition of strong code mobility support in the Holoparadigm environment.

Keywords: Strong code mobility, programming languages, virtual machines, mobile computing.

Sumário

Lista de Abreviaturas

ANSI: *American National Standards Institute*
API: *Application Programming Interface*
COD: *Code on Demand*
CLR *Common Language Runtime*
DNS: *Domain Name System*.
DSM: *Distributed Shared Memory*
FIPA: *Foundation for Intelligent Physical Agents*
GPS: *Global Positioning System*
HNS: *Holo Naming System*
HOLO: *Holoparadigma*
HS: *History Server*
HVM: *Holo Virtual Machine*
IP: *Internet Protocol*
ISAM: *Infra-estrutura de Suporte a Aplicações Móveis Pervasivas*
JADE: *Java Agent Development Framework*
JVM: *Java Virtual Machine*
KS: *Knowledge Source*
MCS: *Mobile Code System*
MA: *Mobile Agent*
NAT: *Network Address Translator*
PDA: *Personal Digital Assistant*
PHOLO: *Pervasive Holoparadigm*
RFID: *Radio Frequency Identification*
REV: *Remote Evaluation*
SELIC: *Servidor de Localização e Informação de Contexto*
SQL: *Structured Query Language*
SNMP: *Simple Network Management Protocol*
TDS: *True Distributed System*
TCP: *Transmission Control Protocol*
UDP: *User Datagram Protocol*
URI: *Uniform Resource Identifier*
XML: *Extended Markup Language*

Lista de Figuras

Lista de Tabelas

1. Introdução

1.1 Contextualização

Atualmente a popularização de dispositivos computacionais cada vez menores e com maior poder computacional está fazendo com que a computação móvel, antes apenas acessível a poucas pessoas, esteja se tornando bastante popular. Dispositivos como PDAs *(Personnal Digital Assistant*), computadores portáteis, celulares estão tornando-se cada vez mais poderosos computacionalmente ao mesmo tempo em que se tornam acessíveis a um maior número de pessoas tornando possível que a computação atue em cada vez mais áreas a um custo cada vez menor. Aliado a este avanço dos dispositivos, tecnologias de software emergem de modo a aproveitar as novas possibilidades motivadas por este avanço. Novos protocolos de rede sem fio, como *Bluetooth* [9] e o IEEE 802.11b/g [46] fazem com que estes dispositivos possam conectar-se de modo a aumentar ainda mais suas funcionalidades.

Esta disseminação da computação, entretanto, não está livre de desafios. Na medida em que os dispositivos evoluem, os desenvolvedores de software devem adaptar-se a esta nova realidade através do desenvolvimento de aplicações que venham a explorar estas novas tecnologias ao mesmo tempo suprindo as necessidades cada vez mais específicas de um número cada vez maior de usuários. À medida que o número de usuários cresce, questões referentes à escalabilidade tornam-se uma preocupação. Uma das propostas que vem ao encontro da solução deste problema é a mobilidade de código. A mobilidade de código propicia o desenvolvimento de programas móveis que tem a habilidade de trocar de dispositivos durante sua execução de modo a suprir a necessidade de uma grande gama de aplicações.

A mobilidade de código não é um conceito novo. O termo código móvel tem sido utilizado na literatura de acordo com as seguintes definições [42]:

- O termo código móvel pode ser utilizado para descrever qualquer programa que pode ser enviado sem modificações para uma coleção heterogênea de dispositivos e executado com uma semântica semelhante em cada um deles;

- Código móvel pode ser visto como uma abordagem onde programas são considerados documentos e consequentemente possam ser acessados, transmitidos e executados como qualquer outro documento;

- Agentes móveis são objetos que contém código e que podem ser transmitidos entre dispositivos participantes de uma arquitetura distribuída.

Neste trabalho, código móvel será visto como um software que é capaz de viajar por uma rede heterogênea e executar automaticamente no dispositivo de destino. Esta caracterização é genérica o suficiente para englobar a maioria das utilizações de código móvel e, ao mesmo tempo precisa o suficiente para destacar as principais características desta técnica. Ela exclui, por exemplo, situações onde um código é carregado de um disco compartilhado ou obtido manualmente através da Internet.

1.2 Definição do Problema

Um ambiente altamente dinâmico composto por dispositivos móveis demanda que aplicações sejam adaptativas por natureza. Neste sentido torna-se interessante a obtenção de novos elementos lógicos em tempo de execução, obtendo de maneira dinâmica as funcionalidades para atender as demandas do ambiente. Estas aplicações devem ser sensíveis ao contexto, ou seja, elas devem adaptar-se as constantes mudanças deste tipo de ambiente de uma maneira transparente para o usuário final.

Outro problema inerente a este tipo de ambiente refere-se à heterogeneidade do mesmo. Uma grande variedade de dispositivos móveis interage utilizando diferentes tipos de rede de comunicação. Conexões temporárias entre dispositivos e perdas repentinas de conexão são uma realidade em um ambiente móvel. Uma aplicação móvel deve aproveitar os recursos disponíveis neste ambiente da melhor forma possível além de lidar com as adversidades impostas. Neste sentido a forma de composição e programação da aplicação torna-se uma questão importante. A maioria das soluções existentes utiliza-se de paradigmas já consagrados para implementar suas soluções. Entretanto estes paradigmas foram concebidos com aplicações que executam em um único nodo em mente. Soluções utilizando estes paradigmas não podem ser diretamente aplicadas em um ambiente

altamente heterogêneo e dinâmico. Neste trabalho esta questão é abordada através da utilização de mobilidade de código.

Através da utilização de mobilidade de código uma forma mais flexível de sistema distribuído é obtida, onde a localização das computações não precisa ser conhecida de antemão. Dentre as principais vantagens deste modelo destacam-se a eficiência, simplicidade e flexibilidade [42].

Estas questões motivam o desenvolvimento de novas abstrações para programação bem como a implementação do respectivo suporte. Neste trabalho uma abstração será proposta através da utilização do Holoparadigma [4] (de maneira abreviada Holo) e materializada através da implementação na HoloVM [23], que é a máquina virtual que executa *byte code* Holo. Atualmente o Holoparadigma oferece uma abstração intuitiva para a modelagem de ambientes móveis. Através de sua utilização é possível criar uma modelagem mais fiel do mundo real. Utilizando diretivas de mobilidade do próprio paradigma é possível manter o ambiente coerente com o mundo real. Esta abordagem aplica-se tanto para ambientes fixos (prédios, salas, computadores, etc.) quanto para elementos móveis (*laptops*, PDAs, celulares, etc.). Até então, o ambiente de execução utilizado não suportava a mobilidade de código para o desenvolvimento de aplicações, suprimindo assim um grande potencial do Holoparadigma.

1.3 Objetivos

Este trabalho tem como objetivo especificar, implementar e validar um sistema de mobilidade de código para o Holoparadigma. Com este objetivo geral em mente, os seguintes objetivos específicos podem ser citados:

- Especificar e implementar um modelo de mobilidade de código;
- Introduzir este modelo no Holoparadigma, através de sua integração com a HoloVM e com o ambiente de execução distribuído;
- Validar este modelo através da implementação de aplicações sintéticas que utilizem mobilidade de código.

1.4 Metodologia

Visando cumprir os objetivos propostos uma metodologia foi definida para o cumprimento das tarefas desta proposta. Inicialmente um estudo de sistemas de código móvel foi realizado de modo a identificar suas principais características. Este estudo visou verificar como a mobilidade de código é disponibilizada por estes sistemas.

Após a realização deste estudo foi realizado um levantamento dos requisitos necessários para a implementação de mobilidade de código no Holoparadigma. Esta etapa englobou um estudo da HoloVM de modo a identificar que partes dela necessitam ser modificadas. A fase seguinte compreendeu a implementação de um protótipo inicial que fornece mobilidade de código para programas que executam *byte code* Holo. Tendo este protótipo implementado, foram desenvolvidas aplicações sintéticas com o objetivo de validar o modelo proposto e também identificar que pontos devem ser melhorados.

1.5 Organização da Dissertação

O restante deste trabalho está estruturado da seguinte forma. No capítulo seguinte será apresentado o conceito de mobilidade de código através da descrição de um *framework* metodológico e terminológico juntamente com paradigmas de desenvolvimento de software que utilizam mobilidade de código. Ainda neste capítulo serão apresentados sistemas que utilizam mobilidade de código. No Capítulo 3 será apresentado o Holoparadigma, juntamente com a HoloVM e o ambiente de execução distribuído. O modelo proposto é apresentado no Capítulo 4. O Capítulo 5 apresenta aspectos específicos de implementação juntamente com resultados iniciais. Aplicações desenvolvidas, e que utilizam o modelo de mobilidade forte de código desenvolvido são apresentadas no Capítulo 6. A conclusão juntamente com trabalhos futuros é apresentada no Capítulo 7.

2. Mobilidade de Código

Atualmente as redes de computadores estão evoluindo de maneira rápida e esta evolução acontece em diferentes frentes. Além do tamanho das redes que cresce rapidamente, por exemplo, a Internet, isso também acontece com a velocidade das conexões. Este aumento no tamanho e desempenho das redes de computadores pode ser visto como uma causa e também como um efeito de um importante fenômeno: as redes estão se tornando ubíquas [16], ou seja, além de serem um recurso acessível a um grande número de pessoas, a conectividade disponibilizada por elas pode ser explorada independente da localização física do usuário. Neste cenário o usuário pode movimentar-se por diferentes locais e ainda manter-se conectado, possibilitando que uma nova gama de aplicações seja desenvolvida visando aproveitar este avanço.

Esta evolução, entretanto não é livre de obstáculos e novos problemas devem ser resolvidos. Com este crescimento acelerado a escalabilidade torna-se uma preocupação. Algoritmos propostos com pequenas redes em mente não podem ser aplicados diretamente a uma rede de escala maior como a Internet. Outro problema surge das conexões sem fio. Nestes ambientes o usuário é livre para se locomover gerando uma mudança dinâmica na topologia da rede, o que pode ocasionar a perda temporária ou permanente da conexão. Outra questão relevante que aparece neste cenário é a difusão de serviços de rede para grandes segmentos da sociedade. Esta difusão requer que os serviços oferecidos possam ser customizados, de modo a suprir as necessidades de diferentes usuários. Outra necessidade, ocasionada pela mudança constante na infra-estrutura de comunicação subjacente, é que as redes sejam flexíveis de modo a se adaptarem de maneira rápida e eficaz. Diversas tentativas de resolver estes problemas baseiam-se em tecnologias bem estabelecidas. Estas tecnologias são adaptadas a esta nova realidade visando fornecer customização e flexibilidade. Muitas destas tecnologias, como CORBA [35] utilizam o paradigma cliente/servidor. Entretanto autores [21] [32] sugerem que uma abordagem utilizando mobilidade de código seja mais adequada a um ambiente desprovido de flexibilidade e customização como o descrito anteriormente. Neste trabalho um modelo que aborda a mobilidade de código é proposto de modo a resolver este problema. Na próxima seção são

apresentados conceitos envolvendo mobilidade de código que serão utilizados no restante do trabalho.

2.1 *Framework* para Mobilidade de Código

A mobilidade de código não é um conceito novo. Diversas tecnologias utilizam implicitamente o conceito de mobilidade de código. Dentre estas se pode citar o *Postscript* [42], que é uma linguagem de descrição de páginas que se destaca por ser uma linguagem de programação baseada em pilha. Ao enviar um documento para uma impressora *Postscript*, o que está sendo enviado é um programa que descreve as páginas a serem impressas. A impressora, ao receber o programa, executa-o de modo a realizar a impressão. Este exemplo demonstra uma das principais motivações da mobilidade de código, pois a descrição algorítmica de uma imagem complexa, por exemplo, pode ser enviada de maneira compacta e genérica para diversas impressoras, independente de sua resolução ou esquema de cores. Outra tecnologia que utiliza mobilidade de código refere-se a banco de dados [18]. O tamanho de uma base de dados pode tornar inviável a sua transmissão por inteiro para uma estação cliente. Neste sentido, qualquer operação em uma base de dados deste tipo deve ser comunicada e executada no servidor. Atualmente a maioria dos bancos de dados comerciais suporta a linguagem SQL (*Structured Query Language*) para acesso a bases de dados. Esta linguagem oferece uma notação compacta para expressar operações complexas em múltiplas relações de uma base de dados.

Informalmente a mobilidade de código pode ser definida como a modificação de maneira dinâmica da "ligação" de fragmentos de código com o local onde eles executam [21]. Devido a grande quantidade de pesquisas ocorrendo paralelamente a respeito de mobilidade de código, um *framework* metodológico e terminológico é sugerido [21]. O restante deste trabalho adota o conceito de Sistema de Código Móvel (*Mobile Code System - MCS*) [21]. Este modelo, apresentado na Figura 2.1, se diferencia de sistemas distribuídos tradicionais (*True Distributed Systems – TDS*) por não fornecer a transparência de localização, ou seja, o software executando é ciente de sua localização juntamente com a localização dos recursos que ele utiliza. O conceito de "ambiente computacional" é introduzido. O ambiente computacional pode ser visto como uma camada de software executando acima do sistema operacional cuja responsabilidade é gerenciar a execução dos

componentes e também dos recursos que estes utilizam. Estes recursos podem estar localizados no mesmo “ambiente computacional” ou em um ambiente remoto, sendo acessados através da rede.

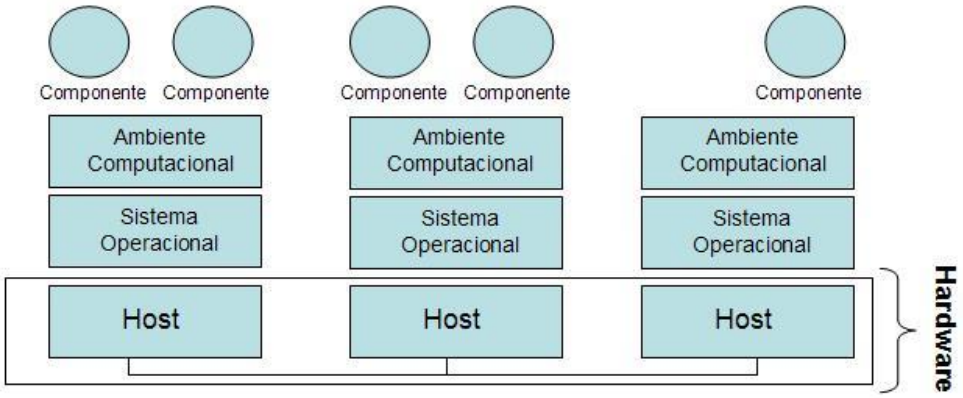

Figura 2.1 - Modelo de um sistema de código móvel

Dentre as principais características destes sistemas destacam-se a escalabilidade e a maior autonomia oferecida ao desenvolvedor. A escalabilidade é ressaltada, pois neste modelo a mobilidade de código é explorada em nível de Internet. A autonomia ocorre, pois é de responsabilidade do programador definir quando a mobilidade irá ocorrer, ao contrário de sistemas distribuídos tradicionais onde a mobilidade ocorre de maneira transparente ao desenvolvedor.

Neste modelo cada ambiente computacional possui uma ou mais unidades de execução juntamente com recursos necessários para a execução. Uma unidade de execução pode ser vista como a representação de um fluxo de execução como, por exemplo, um processo Unix ou uma *thread* em um ambiente *multithreaded*. Os recursos podem estar localizados no próprio ambiente computacional ou em um ambiente computacional localizado em outro *host*. Um recurso localizado em um *host* remoto é acessado através da rede.

Cada unidade de execução é composta por um segmento de código, um espaço de dados juntamente com o estado de execução. A estrutura de uma unidade de execução é descrita na Figura 2.2.

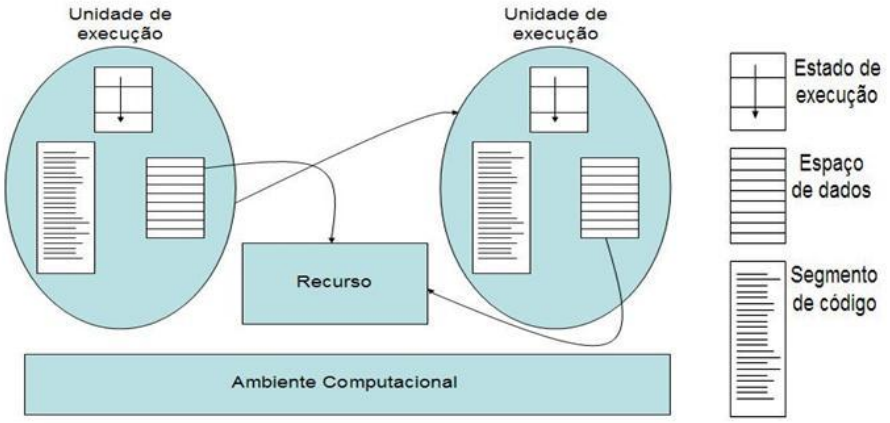

Figura 2.2 - Estrutura interna de uma unidade de execução

Como mostrado na Figura 2.2, um recurso local a um ambiente computacional pode ser compartilhado entre unidades de execução. Diferentes maneiras, apresentadas mais adiante, são propostas para gerenciar os recursos utilizados por um ambiente computacional.

Na literatura são apresentados dois tipos de mobilidade de código. A mobilidade forte e a mobilidade fraca [21] [32] [24]. Na mobilidade forte, o código de uma unidade de execução é movido juntamente com seu estado de execução para um ambiente computacional diferente. Na mobilidade fraca, somente o código é movido e, em alguns casos, dados necessários para inicialização da execução no ambiente remoto são movidos junto. O suporte à mobilidade forte pode ocorrer através de duas abordagens: migração e clonagem remota. Na migração a execução em uma unidade de execução é interrompida, seu estado de execução é armazenado juntamente com o código e então migrado para outro ambiente computacional. Na abordagem que utiliza clonagem remota, a unidade de execução continua executando em seu ambiente computacional enquanto ela é clonada para outro ambiente computacional. Tanto a abordagem que utiliza migração quanto a abordagem que utiliza clonagem remota podem ser pró-ativas ou reativas. Em uma abordagem pró-ativa, é a própria unidade de execução que decide quando a migração ou a clonagem remota irá ocorrer. Contrastando com a abordagem pró-ativa, na abordagem reativa a decisão de quando a migração ou a clonagem remota irá ocorrer não é feita pela unidade de execução. Nesta abordagem o que gera a mobilidade é outra unidade de execução que geralmente possui algum tipo de relacionamento com a unidade de execução

que contém o código a ser movido. Um cenário possível para uma abordagem reativa seria uma unidade de execução atuando como uma entidade gerenciadora das demais unidades de execução.

Tanto a abordagem pró-ativa quanto a abordagem reativa podem executar a mobilidade de código de maneira síncrona ou assíncrona. Em uma abordagem síncrona o código é executado no momento de sua chegada ao ambiente de destino. Em uma abordagem assíncrona o código pode vir a ser executado somente quando alguma condição for satisfeita.

Mecanismos que suportam mobilidade de código fraca possuem a capacidade de transferir código entre ambientes computacionais ligando o código movido de maneira dinâmica no novo ambiente computacional ou fazendo com que o código movido se torne o segmento de código do novo ambiente computacional. Tais mecanismos podem ser classificados de acordo com a direção do código sendo transferido, a natureza do código sendo movido, o tipo de sincronização envolvido e o momento em que o código movido será executado no ambiente computacional de destino. Quanto à direção do código sendo transferido, uma unidade de execução pode executar uma busca de código ou executar o envio de código para outro ambiente computacional. O código pode ainda ser transferido de maneira isolada, neste caso instanciando uma nova unidade de execução no ambiente de destino, ou pode ser enviado apenas na forma de fragmento que terá que ser ligado para uma execução no ambiente computacional de destino. Estes mecanismos ainda podem ser síncronos ou assíncronos. Em uma abordagem síncrona a unidade de execução que requisita o código suspende sua execução até que o código requisitado esteja disponível. Alternativamente, na abordagem assíncrona, a unidade de execução não suspende a execução.

Ao migrar de um ambiente computacional para outro, uma unidade de execução deve rearranjar seu espaço de dados a fim de continuar sua execução no ambiente computacional de destino. Diferentes soluções são propostas para realizar esta nova configuração. Algumas envolvem a remoção da ligação com o recurso envolvido enquanto outras envolvem a criação de uma referência através da rede para o ambiente computacional onde se encontra o recurso. Dependendo ainda do tipo do recurso, este pode

ser migrado juntamente com a unidade de execução para o novo ambiente computacional. Outra possibilidade ainda é a remoção do recurso sendo utilizado caso nenhuma das alternativas seja possível de ser aplicada. A Figura 2.3 mostra na forma de uma taxonomia os conceitos apresentados até agora.

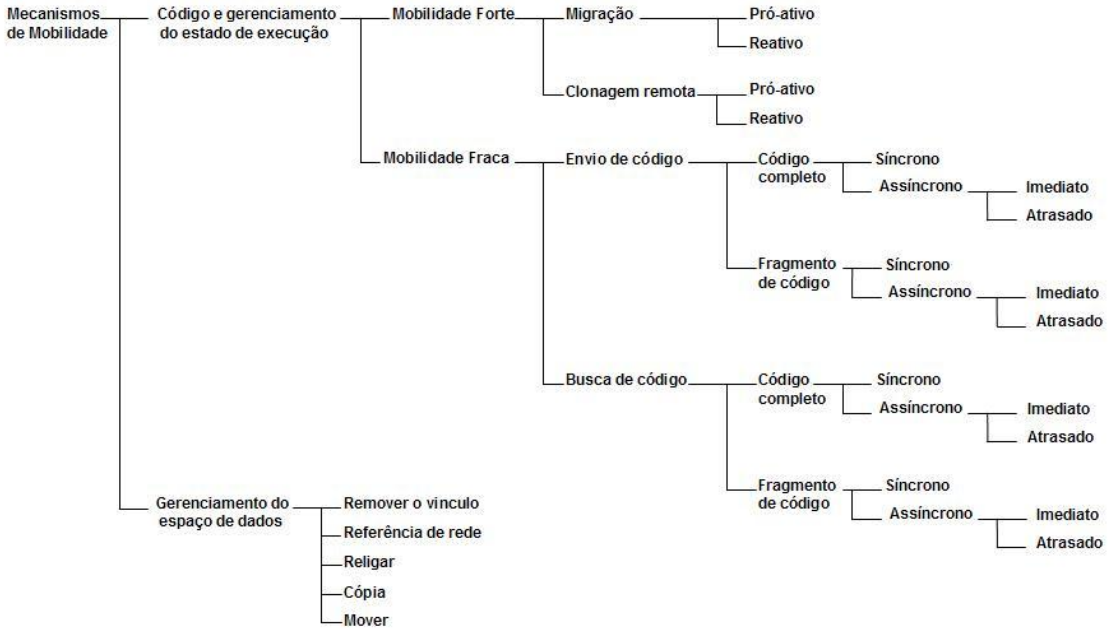

Figura 2.3 - Taxonomia da Mobilidade de Código [21]

A mobilidade de código, embora não sendo um conceito novo, continua sendo bastante utilizada em diversas áreas. A crescente popularização de dispositivos cada vez menores e com maior poder computacional vem abrindo um leque de possibilidades em termos de computação até pouco tempo atrás não imaginada. Dentre as principais utilizações da mobilidade de código, pode-se citar:

- Distribuição de carga: uma unidade de execução pode migrar de um ambiente computacional que está sobrecarregado para um ambiente que esteja ocioso visando um melhor aproveitamento;

- Tolerância à falhas: uma unidade de execução pode migrar de um ambiente computacional que possa estar tendo problemas e continuar sua execução em um ambiente sem falhas;

- Compartilhamento de recursos: mesmo não sendo disponibilizado em um ambiente computacional, um recurso específico pode ser aproveitado por uma unidade de execução através da migração da mesma para um ambiente computacional que possua o recurso requisitado. Como exemplo de recurso específico pode-se citar uma impressora ou uma base de dados;

- Localidade no acesso aos dados: ao necessitar de um dado uma unidade de execução pode se mover para o ambiente computacional que possua o dado;

- Computação móvel: uma unidade de execução pode se mover de um computador de mesa para um computador portátil como um *notebook* ou um PDA. Este tipo de mobilidade se aplica a semântica siga-me da computação ubíqua [3]. Uma das formas de se obter um sistema ubíquo é através da implementação de plataformas virtuais, como máquinas virtuais, que possibilitem a mobilidade de código. Esta mobilidade de código tornaria possível que o código seguisse o usuário para onde ele fosse;

- Customização de serviços: diferentes usuários requerem diferentes tipos de serviços disponibilizados. Na medida em que o número de usuários cresce, uma demanda maior por serviços personalizados se faz necessária. Neste sentido a mobilidade de código pode atuar como um agente facilitador neste processo;

- Inserção dinâmica de novos protocolos de procura de *hosts* ou dispositivos: dispositivos móveis, apesar de um poder computacional cada vez mais elevado, muitas vezes possuem capacidades limitadas de armazenamento. Neste sentido é interessante que estes dispositivos possuam instalado somente os componentes extremamente necessários para o seu funcionamento. Ao necessitar de um novo componente o dispositivo o obteria de maneira dinâmica. Ao final de sua execução o componente pode inclusive descartar este componente liberando assim mais recursos para outras aplicações.

2.2 Paradigmas de Projeto de Mobilidade de Código

Nesta seção serão apresentados paradigmas de desenvolvimento de aplicações distribuídas que utilizam mobilidade de código. Questões específicas de linguagens ou tecnologias de mobilidade de código serão abstraídas. A intenção é apresentar, em um nível alto de abstração, questões referentes ao projeto destas aplicações distribuídas de modo a oferecer diretivas que possam vir a ser utilizadas no desenvolvimento de aplicações distribuídas que utilizem mobilidade de código. Assim como os benefícios da orientação a objeto como maior legibilidade e reutilização podem ser aplicados utilizando-se uma linguagem não orientada a objetos, os paradigmas apresentados nesta seção podem ser aplicados independente da linguagem ou tecnologia de mobilidade de código utilizada.

Na fase de projeto de uma aplicação distribuída, normalmente questões referentes à interação entre os componentes são consideradas independentes da localização deles. A localização, nestes casos, é tratada apenas como uma questão de implementação. Em alguns casos, detalhes de localização são definidos pelo programador na fase de implementação. Em outros casos, a localização é abstraída por uma camada de *middleware*. CORBA, por exemplo, é um *middleware* que esconde de maneira intencional do programador a localização dos componentes do sistema. Neste *middleware* não há distinção entre componentes situados no mesmo ambiente computacional e componentes situados em ambientes diferentes. Esta, entretanto, não é a única abordagem possível para projetar uma aplicação distribuída. Existem casos onde o conceito de localização, recursos e migração de componentes para outros ambientes computacionais devem ser levados em consideração já na fase de projeto. Em alguns casos a interação entre componentes que residem no mesmo ambiente computacional será muito diferente da interação entre componentes que residem em ambientes diferentes se forem levadas em conta questões como latência da rede, falhas parciais no *link* de comunicação, acesso a memória, concorrência e até mesmo diferentes capacidades computacionais entre os ambientes computacionais envolvidos. A abstração destas questões pode levar a resultados inesperados quanto ao desempenho e confiabilidade do sistema [27].

No restante desta seção, serão abordados paradigmas de projeto aplicados a uma classe de aplicações nas quais os conceitos de mobilidade e localização têm um papel

fundamental que podem vir a modificar a estrutura conceitual da aplicação concebida durante a fase de projeto [24]. Antes de entrar em detalhes nos paradigmas propriamente ditos serão introduzidas algumas abstrações utilizadas no restante da seção para definir os paradigmas:

- Componentes: são considerados os elementos da arquitetura. Podem ser subdivididos em recursos e unidades de execução. Recursos compreendem elementos representando dados passivos ou dispositivos físicos como arquivos, impressoras ou ainda uma interface de rede. Uma unidade de execução corresponde a um fluxo de execução como uma *thread* ou um processo. São caracterizados por possuírem um estado que contém dados privados, o estado de sua execução bem como referências para os recursos utilizados. Podem ser vistos também como o *know-how* necessário para a execução do serviço;

- Interações: são eventos que envolvem dois ou mais componentes. Uma mensagem trocada entre duas unidades de execução, por exemplo, é uma interação;

- Locais: são ambientes computacionais. Eles hospedam componentes e provêem suporte para a execução de unidades de execução.

Atualmente o paradigma mais utilizado em aplicações distribuídas é o cliente-servidor. Neste modelo o cliente envia uma requisição de serviço ao servidor, este processa a requisição enviando de volta para o cliente somente a resposta do serviço. Tanto a computação quanto os recursos necessários para executá-la estão localizados no servidor.

Neste contexto, três paradigmas [21] [24] que exploram mobilidade de código e que estendem o paradigma cliente-servidor serão apresentados. Estes paradigmas serão diferenciados de acordo com a localização dos componentes antes e depois do código ser executado, qual componente executa o código e onde a computação é realizada. Os paradigmas apresentados serão os seguintes: *remote evaluation* (REV), *code on demand* (COD) e *mobile agent* (MA) . Com o objetivo de obter um melhor entendimento destes paradigmas será considerado um cenário onde um ambiente computacional **A** localizado no

local **La** necessita do resultado de um serviço. Neste cenário ainda existe outro ambiente computacional **B** localizado no local **Lb** que estará envolvido no processo de execução do serviço.

2.2.1 *Remote Evaluation* (REV)

Neste paradigma um componente possui o *know-how* para executar o serviço, mas não possui os recursos necessários para a execução dele. Desta maneira ele envia o seu *know-how* para um componente situado em um ambiente remoto. Este componente executa o serviço baseado no *know-how* recebido e envia a resposta para o componente que iniciou a transação. Após a execução do serviço, o componente que o executou possui além dos recursos que ele já possuía o *know-how* enviado pelo componente que requisitou o serviço. Dentro do contexto do cenário descrito anteriormente, o componente **A**, localizado no local **La** envia o código a ser executado para o componente **B** no local **Lb**. Após a execução, o componente **B** possui além dos recursos necessários para a execução do serviço, o código representando o *know-how* recebido do componente **A**.

2.2.2 *Code on Demand* (COD)

Neste paradigma o componente possui os recursos necessários para a execução do serviço, mas não possui o *know-how* necessário para a execução dele. Sendo assim ele, através de uma interação com outro componente, requisita que este o envie o *know-how* necessário para a execução do serviço. Após a execução do serviço o componente que o requisitou possui os recursos e o *know-how* enviado pelo outro componente. No contexto do cenário exemplo, o componente **A** localizado no local **La** possui apenas os recursos necessários, mas não possui o *know-how*. Sabendo que o componente **B** possui o *know-how* necessário para a execução do serviço, o componente **A** requisita que este o envie. Após a execução do serviço o componente **A** possui além dos recursos o *know-how* recebido do componente **B**.

2.2.3 *Mobile Agent* (MA)

Neste paradigma o componente possui o *know-how* necessário para a execução do serviço e possui apenas alguns recursos. Sendo assim o componente executa em seu local

até quando puder. Quando um recurso não estiver disponível, o componente migra então para outro local onde ele possa continuar sua execução. Fazendo uma analogia com o cenário de exemplo, o componente **A** inicia sua execução em seu ambiente e, em certo ponto da execução ele migra para outro local mais apropriado para continuar a sua execução. Neste paradigma, diferentemente dos paradigmas REV e COD, não é somente o código que migra. Ao migrar de um local para outro o componente pode levar junto consigo resultados intermediários computados ainda no local de partida.

2.2.4 Discussão

Tendo apresentado estes paradigmas de desenvolvimento, sumarizados na Tabela 2.1, cabe ressaltar que estes não são os únicos. Outros paradigmas de desenvolvimento podem ser utilizados em aplicações que utilizam mobilidade de código. Ao se projetar uma aplicação que virá a utilizar mobilidade de código, diversas questões devem ser analisadas de modo a escolher o melhor paradigma. Até mesmo mais de um paradigma pode ser utilizado na mesma aplicação. Dependendo dos requisitos da aplicação um paradigma pode ser mais adequado que outro. Se uma aplicação, por exemplo, necessita utilizar pouca largura de banda, o paradigma MA em uma primeira análise parece ser o mais adequado. Da mesma maneira se uma aplicação necessita executar em um ambiente com memória limitada, o paradigma COD é uma boa opção dado que o código é obtido somente no momento de sua execução podendo inclusive vir a ser descartado após a mesma, liberando os recursos utilizados.

Todos os paradigmas apresentados explicitam o conceito de local. Esta abstração é introduzida no nível de arquitetura a fim de levar em conta na hora da modelagem da aplicação a localização dos componentes. A maioria dos paradigmas utilizados atualmente, ao contrário dos paradigmas apresentados, não explora o conceito de mobilidade não podendo assim aproveitar os benefícios oferecidos.

Tabela 2.1 - Resumo dos paradigmas de desenvolvimento de mobilidade de código [24]

Paradigma	Antes		Depois	
	La	Lb	La	Lb
Cliente Servidor	A	B Recursos *Know-how*	A	B Recursos *Know-how*
Execução Remota	A *Know-how*	B Recursos	A	B *Know-how* Recursos
Código sob Demanda	A Recursos	B *Know-how*	A Recursos *Know-how*	B
Agente Móvel	A *Know-how*	Recursos	-	A Recursos *Know-how*

Uma vez escolhido o paradigma a ser utilizado, outra escolha que os desenvolvedores devem fazer é qual tecnologia deverá ser utilizada na implementação do sistema. Mesmo as tecnologias disponíveis sendo na maioria das vezes ortogonais com relação ao paradigma escolhido, muitas vezes tecnologias específicas são mais apropriadas a determinados paradigmas. Como exemplo pode-se citar a utilização de um sistema de código móvel baseado em mobilidade fraca para programar um sistema utilizando o paradigma de agente móvel (MA). Neste cenário, ao mover um componente de um ambiente computacional para outro, é de responsabilidade do programador criar as estruturas de dados necessárias para manter o estado de execução do componente (agente) durante a migração. Neste sentido o programador é quem teria que armazenar, enviar e por fim restaurar o estado corrente de execução do agente no ambiente destino. Se, ao invés de utilizar um sistema de código móvel baseado em mobilidade fraca, fosse utilizado um sistema baseado em mobilidade forte, a migração do componente se restringiria a apenas uma instrução.

2.3 Sistemas de Código Móvel

Tendo apresentado nas seções anteriores os tipos de mobilidade de código juntamente com um modelo conceitual, nesta seção serão apresentados sistemas que

utilizam mobilidade de código. Estes sistemas serão classificados de acordo com os conceitos apresentados anteriormente. Em alguns destes sistemas, mais precisamente os sistemas ubíquos, a mobilidade de código é vista apenas como uma pequena parte do sistema. Embora não sendo o objetivo fim destes sistemas, ela ainda assim tem um papel fundamental na obtenção das funcionalidades oferecidas.

2.3.1 *Voyager*

Desenvolvido pela empresa Recursion Software Inc, Voyager [44] é uma plataforma que integra de maneira simples o desenvolvimento de agentes móveis a aplicações distribuídas. Voyager oferece suporte a objetos móveis, tornando possível que objetos se movam até o lugar onde tenham que realizar suas tarefas e retornem com os resultados. Além deste suporte, Voyager também oferece um *framework* para agentes autônomos. Uma de suas principais características é a facilidade de utilização, simplificando o desenvolvimento de aplicações que utilizam de alguma forma a mobilidade de código, seja ela forte ou fraca.

O *framework* disponibilizado pela plataforma Voyager é implementado na linguagem Java e permite que, através de uma única mensagem, um objeto seja movido de uma máquina virtual para outra mesmo se este objeto esteja sendo utilizado por outros clientes no momento. Mesmo com uma falha parcial ou total da rede, agentes Voyager podem continuar sua execução em clientes remotos. Adicionalmente agentes Voyager podem implantar código em sistemas que estejam executando sem a necessidade de parar a execução destes. Mais informações podem ser encontradas em [44] e [34].

2.3.2 Java

Desenvolvida pela empresa Sun Microsystems, Java [25] é uma linguagem de programação orientada a objetos com construções semelhantes ao C++. Foi concebida desde o início para ser uma linguagem simples, portável, de fácil aprendizado e de propósito geral. Atualmente a linguagem Java encontra-se na versão 1.5 e desde seu lançamento em 1995 muitas funcionalidades foram acrescentadas à linguagem.

O compilador Java traduz programas escritos em Java para uma linguagem intermediária, independente de plataforma chamada Java *byte code*. Este *byte code* por sua vez é interpretado pela máquina virtual Java JVM. Uma das características mais fortes da linguagem Java é sua portabilidade. Devido a sua arquitetura, um programa compilado em Java pode executar em qualquer sistema operacional que possua uma máquina virtual implementada. Sendo assim um programa escrito em Java, uma vez compilado, pode ser executado por qualquer máquina virtual Java.

A máquina virtual Java tem a capacidade de carregar, em tempo de execução, o código (*byte code*) necessário para execução. Quando a máquina virtual está executando um *byte code* e encontra um nome de classe não resolvido, ela pode carregar de maneira dinâmica (através do *class loader*) o código necessário. Este código pode estar localizado na mesma máquina ou em uma máquina remota. Adicionalmente, o carregamento dinâmico de código pode ser explicitado pela aplicação. A execução do código carregado pode ser imediata ou atrasada. Em ambos os casos o código é sempre executado do início, ou seja, não existe tratamento para o estado de execução, caracterizando assim a mobilidade de código oferecida como fraca.

Encaixando a linguagem Java no modelo de mobilidade de código descrito na seção anterior, tem-se que o ambiente computacional é a JVM e a unidade de execução é o código Java que ela executa.

2.3.3 Java Aglets

Aglets [30] é um SDK (*Software Development Kit*) desenvolvido pelo IBM Tokyo Research Laboratory no Japão. É um sistema de agentes móveis desenvolvido em Java onde os agentes são denominados aglets. Cada agente neste sistema é uma *thread* Java executando em uma máquina virtual Java. O ambiente de execução é oferecido através de um componente, chamado Tahiti server [20]. É este componente quem suporta a execução, disponibiliza mecanismos para a mobilidade dos agentes e implementa mecanismos de segurança.

A mobilidade oferecida por este sistema é fraca, pois o estado de execução dos agentes não é movido junto com eles. Ao mover-se de um ambiente computacional para

outro o agente suspende a *thread* que ele está executando, e migra para outro ambiente computacional sem levar consigo seu estado de execução, reiniciando sua execução ao chegar ao ambiente de destino [12].

2.3.4 JavaGo

Este sistema foi desenvolvido em Java e oferece suporte à mobilidade forte. Para oferecer a mobilidade forte JavaGo [39] utiliza um pré-processador que modifica em tempo de compilação o código fonte gerado. As modificações realizadas pelo pré-processador possibilitam ao JavaGo capturar o estado de execução do agente antes de enviá-lo a um computador remoto.

Estas transformações possibilitam que o programador possa escolher inclusive que parte da pilha de execução deve ser migrada. A migração do agente ocorre através de uma única instrução, o que simplifica em muito o desenvolvimento de programas que o utilizam.

Outro sistema, baseado no JavaGo, denominado JavaGoX [38], a fim de também oferecer suporte à mobilidade forte, realiza uma transformação no *byte code* gerado pelo compilador.

2.3.5 JADE

JADE [19] (*Java Agent Development Framework*) é um *framework* desenvolvido e suportado pela CSELT da Universidade de Parma na Itália que objetiva facilitar o desenvolvimento de aplicações multi-agentes. JADE pode ser considerado um *middleware* que implementa uma plataforma para o desenvolvimento de agentes. Sua implementação segue as especificações da FIPA[1] (*Foundation for Intelligent Physical Agents*). Em JADE toda a comunicação entre os agentes ocorre através de troca de mensagens.

JADE foi escrito em Java devido às características particulares da linguagem [6], principalmente pela facilidade de utilização da linguagem em ambientes heterogêneos distribuídos. JADE disponibiliza tanto classes prontas para serem utilizadas quanto

[1] http://www.fipa.org

interfaces abstratas as quais o desenvolvedor pode utilizar como base de implementação de acordo com suas necessidades.

Utilizando o modelo de mobilidade de código descrito na seção anterior tem-se que JADE encaixa-se no paradigma de agentes móveis. Adicionalmente, em JADE, o ambiente de execução é dado por uma instância de uma máquina virtual Java e cada agente é uma *thread* Java. Na sua versão mais recente, modificações foram realizadas no *framework* visando aumentar o grau de segurança dos sistemas desenvolvidos [20]. JADE suporta tanto a migração quanto a clonagem de agentes.

2.3.6 Jini

Jini [28] [48] é uma tecnologia desenvolvida pela empresa Sun Microsystems que assume uma rede em constante mudança tanto em termos dos componentes que compõe a rede quanto na maneira como eles interagem. Esta tecnologia permite que desenvolvedores criem serviços centrados na rede.

A tecnologia Jini é desenvolvida em Java e a mobilidade de código acontece quando um objeto necessita de um determinado serviço. Com o objetivo de adquirir este serviço o objeto realiza uma busca na rede e o serviço é enviado para o objeto na forma de *byte code* Java. Sendo assim, a principal funcionalidade da mobilidade de código na tecnologia Jini é a customização nos serviços oferecidos. A fim de ser localizado por clientes, o serviço registra-se em um *lookup server* que é onde ele será localizado.

A tecnologia Jini, em conjunto com a capacidade da linguagem Java de mover código de maneira segura possibilita que o sistema represente uma forma automática de rede onde serviços e clientes podem ser adicionados ou removidos a qualquer momento. Da mesma maneira serviços existentes podem ser modificados ou incrementados de maneira simples.

2.3.7 µCode

O sistema µCode [37] foi desenvolvido a partir do conhecimento agregado pela construção de aplicações na área de gerenciamento de redes baseadas em agentes móveis.

Este sistema disponibiliza mobilidade fraca. Ao migrar de um ambiente para outro um agente preserva seus atributos, mas seu estado de execução é perdido. Cada agente neste sistema é uma *thread* Java.

Além da semântica reativa de instanciação de classes disponibilizada pela máquina virtual Java, este sistema ainda oferece uma semântica pró-ativa onde as classes são previamente enviadas ao nodo de destino antes de serem efetivamente requisitadas pelo *class loader* daquele nodo. Esta semântica tem por objetivo contemplar os casos em que o *link* de conexão entre origem e destino não é permanente, característica muito comum em dispositivos móveis como PDAs e *laptops*.

A unidade de migração neste sistema é dada por um grupo de classes. Uma API específica é oferecida para a construção de grupos. Classes podem ser adicionadas a um grupo individualmente ou de forma coletiva através da resolução de dependências. Cada grupo possui duas classes especiais: *Handler* que é responsável pelo desempacotamento das classes que compõe o grupo no nodo destino e *Root* que é responsável por iniciar a execução das *threads*.

O contexto de execução de um nodo é encapsulado por um servidor denominado µServer. Cada um destes servidores possui um espaço de classes compartilhado no qual a aplicação pode publicar suas classes possibilitando que demais nodos do sistema possam realizar instalações pró-ativas ou reativas das classes publicadas

2.3.8 Obliq

Obliq [42] [11] é uma linguagem interpretada, fracamente tipada, baseada em objetos desenvolvida no DEC System Research Center. Obliq possibilita a execução remota de procedimentos através de *execution engines* que fazem o papel do ambiente computacional. A unidade de execução em Obliq é representada por uma *thread*. Esta unidade de execução pode requisitar a execução remota de um procedimento. Ao efetuar esta requisição a unidade de execução suspende sua execução e envia o código para o ambiente remoto. Ao chegar ao ambiente remoto uma nova unidade de execução é instanciada para executar o código enviado. A unidade de execução que enviou o código

continua suspensa até que a execução remota termine. Desta maneira a mobilidade oferecida por Obliq é fraca com suporte ao envio de código de maneira síncrona.

Objetos em Obliq são considerados recursos fixos transferíveis. Durante todo seu ciclo de vida eles são ligados ao ambiente computacional que os criou. Mesmo havendo uma migração de um objeto para outro ambiente computacional, este continua acessando os recursos de seu ambiente original através de referências de rede. Estas referências são criadas e tratadas automaticamente no momento de uma migração.

2.3.9 .NET

Com conceitos similares à plataforma Java, .NET [15] é uma plataforma de software que disponibiliza diversas funcionalidades para o desenvolvimento de aplicações distribuídas. No núcleo da plataforma .NET encontra-se o .NET *framework*. Este *framework* é quem provê a estrutura básica para o resto da plataforma e também unifica o modelo de programação entre as diversas linguagens suportadas. Além da estrutura básica, o .NET *framework* disponibiliza também uma máquina virtual responsável pelos serviços de gerenciamento de memória, gerenciamento de *threads*, compilação bem como gerenciamento de código fonte. Esta máquina virtual também é chamada de CLR (*Common Language Runtime*). Além dos serviços citados, o CLR também é quem oferece a interoperabilidade entre as linguagens de programação que compõe a plataforma .NET através da definição de uma linguagem intermediária chamada MSIL (*Microsoft Intermediate Language*) e de um formato único de arquivo executável. Todas as linguagens que compõem a plataforma geram o executável de acordo com este formato. Desta maneira arquivos executáveis gerados por diferentes linguagens podem ser integrados facilmente. A linguagem MSIL é similar ao *byte code* Java pois é uma linguagem baseada em pilha com um rico conjunto de instruções.

Outro aspecto importante da plataforma .NET é que, assim como na linguagem Java, o CLR disponibiliza junto consigo uma biblioteca com diversas funções para tratamento de janelas em ambientes gráficos, integração com diversos bancos de dados, entrada e saída, *threads*, suporte a XML entre outras.

O sucesso da linguagem Java no desenvolvimento de sistemas que utilizam mobilidade de código se deve principalmente às funcionalidades que disponibilizam blocos básicos de construção para este tipo de sistema. A plataforma. NET, da mesma maneira, disponibiliza funcionalidades semelhantes tais como suporte à concorrência, serialização de objetos, carregamento de código e reflexão. Sendo assim, o tipo de mobilidade de código suportado pela plataforma .NET é similar à mobilidade de código suportada pela linguagem Java. Esta mobilidade é considerada fraca, pois o estado de execução não é mantido na ocorrência de uma migração de um objeto.

Atualmente, em comparação com a linguagem Java, não são encontrados muitos sistemas que utilizam mobilidade de código baseados na plataforma .NET. A plataforma .NET assemelha-se em diversos pontos com o sistema μCode descrito na seção 2.3.7. No entanto alguns pontos específicos ainda podem ser explorados na plataforma .NET [15]. Um ponto que chama a atenção é a possibilidade de implementação de mobilidade forte através da utilização de meta dados que podem ser embutidos no arquivo executável. Outro trabalho envolvendo mobilidade de código utilizando .NET pode ser encontrado em [33].

2.3.10 LIME

LIME [1] é um *middleware* voltado ao desenvolvimento de aplicações que requerem mobilidade física de hosts, mobilidade lógica de agentes ou ambos. O projeto de LIME é inspirado no fato de que um dos maiores problemas ao se projetar uma aplicação distribuída que utiliza mobilidade de código pode ser visto como um problema de coordenação [22]. Adicionalmente os autores afirmam que para lidar com este tipo de problema, a disponibilidade de abstrações para lidar e explorar um contexto que muda de maneira dinâmica é fundamental.

A perspectiva de coordenação adotada por LIME é baseada no modelo proposto em [14], onde um espaço de tuplas global e persistente é acessível por todas as entidades que compõem o sistema. Em LIME este espaço de tuplas é redefinido de modo que cada agente móvel possua múltiplos espaços de tuplas compartilhados com os demais agentes. Este compartilhamento é obtido estendendo-se o espaço de tuplas de cada agente de modo a armazenar as tuplas dos demais agentes.

O conjunto de tuplas compartilhadas muda durante o tempo devido a ações dos agentes nas mesmas. Outra razão para uma possível modificação no espaço de tuplas é dada pela entrada ou saída de novos agentes neste espaço. O resultado desta abordagem é um contexto gerenciável de maneira transparente expressado através das modificações ocorridas no espaço de tuplas percebido pelos agentes. Desta maneira o comportamento dos agentes é alterado tanto pela disponibilidade de novos dados quanto pela reação a mudanças no contexto.

É interessante notar que a mobilidade não é manipulada diretamente em LIME. Não existem construções para o disparo de mobilidade, seja ela de agentes ou de hosts. Ao invés disso, o efeito da migração é obtido de maneira indireta através das mudanças observadas no contexto do ambiente. Esta decisão de projeto mantém o modelo mais genérico e, ao mesmo tempo, permite diferentes instanciações do modelo baseado em diferentes noções de conectividade.

2.4 Sistemas Ubíquos

Embora não sejam sistemas de mobilidade de código propriamente ditos, cabe ressaltar que a mobilidade de código é uma questão presente em diversos sistemas ubíquos. A computação ubíqua pode utilizar os três paradigmas de mobilidade de código descritos na seção 2.2 para a implementação da semântica siga-me (*follow me*), também descrita em [3].

2.4.1 Aura

Desenvolvido na Universidade de Carnegie Mellon, Aura [41] é uma arquitetura que tem como característica chave ser orientada as tarefas dos usuários. Entre os desafios centrais da arquitetura deste sistema encontram-se dois objetivos: maximizar a utilização de recursos disponíveis e minimizar a distração do usuário.

Este sistema utiliza o conceito de que o usuário possui uma aura pessoal que se encarrega de obter os recursos necessários a partir do ambiente para que o usuário consiga executar suas tarefas. Entre exemplos de possíveis tarefas pode-se citar a escrita de um artigo, a preparação de uma apresentação ou até mesmo a compra de uma casa.

A fim de disponibilizar esta funcionalidade a arquitetura do sistema Aura necessita de funcionalidades que devem ser implementadas tanto em nível de sistema quanto em nível de aplicação. Nesta arquitetura é apresentado o conceito de *placeholder* que auxilia na tarefa de obter o maior número de informações possível a respeito do usuário. Entre as informações que ele armazena podem estar a aplicação que o usuário está executando, informações pessoais e intenções.

A arquitetura do Aura, apresentada na Figura 2.4, é composta por quatro tipos de componentes:

- *Task Manager*, também chamado de *Prism*, responsável por personificar o conceito de aura pessoal do usuário. Este componente é responsável pelo gerenciamento de tarefas como a mudança de ambiente por parte do usuário, mudança do próprio ambiente, mudança de tarefa e mudança de contexto;

- *Context Observer*, responsável por obter informações relevantes a respeito do contexto físico e reportá-las para o *Prism* e para o *Environment Manager*;

- *Environment Manager*, considerado o ponto de entrada do sistema. Este componente sabe quais entidades estão disponíveis no sistema, que serviços elas oferecem e onde podem ser encontradas. Este componente também possui mecanismos que implementam acesso distribuído a arquivos;

- *Suppliers*, responsáveis por prover os serviços abstratos que compõem as tarefas, tais como edição de texto, vídeo, etc.

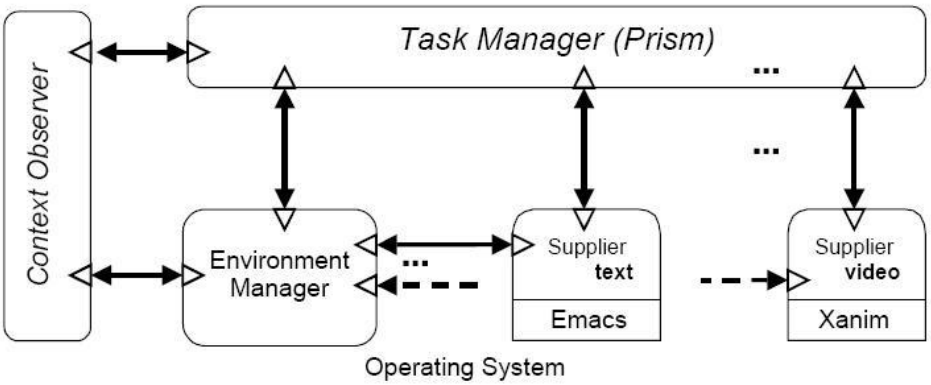

Figura 2.4 - Arquitetura do Aura [41]

A comunicação entre estes componentes ocorre através de um componente chamado *Connector*. Diferentes tipos de *connectors* são disponibilizados para comunicação entre os elementos bem como *connectors* específicos dependendo do tipo de utilização.

A mobilidade de código em Aura se dá através dos componentes descritos anteriormente. Em um exemplo hipotético, um usuário está editando um texto em sua casa e decide ir para seu escritório. Ao perceber a saída do usuário, Aura automaticamente salva seu trabalho, mantendo os aplicativos envolvidos na seção de trabalho do usuário no mesmo estado em que eles estavam no momento em que ele saiu de casa. Caso o usuário estivesse assistindo a um vídeo por exemplo, ao chegar ao seu escritório encontraria o vídeo parado no mesmo instante de tempo. Os arquivos sendo utilizados pelo usuário também seriam enviados para o ambiente do escritório. Todos os recursos necessários para que isto aconteça são tratados por Aura, não necessitando nenhuma intervenção por parte do usuário.

2.4.2 One.World

One.world [26] é uma arquitetura que disponibiliza um *framework* abrangente e integrado para o desenvolvimento de aplicações pervasivas. O principal objetivo é facilitar o desenvolvimento de aplicações que se adaptam de maneira automática a ambientes em constante mudança. A fim de facilitar esta adaptação, one.world oferece aos desenvolvedores serviços que auxiliam no gerenciamento destas mudanças.

Todos os dados em one.world são representados por tuplas, a fim de definir um modelo de dados comum para todas as aplicações objetivando simplificar o compartilhamento de dados. Este modelo de dados comum facilita a tarefa de uma aplicação ao procurar por algum dado específico no ambiente.

Quatro serviços fundamentais são oferecidos em one.world: uma máquina virtual, serviço de tuplas, eventos assíncronos e ambientes. A Figura 2.5 mostra a arquitetura de one.world onde aparecem ambientes representados por retângulos juntamente com componentes e espaços de tuplas.

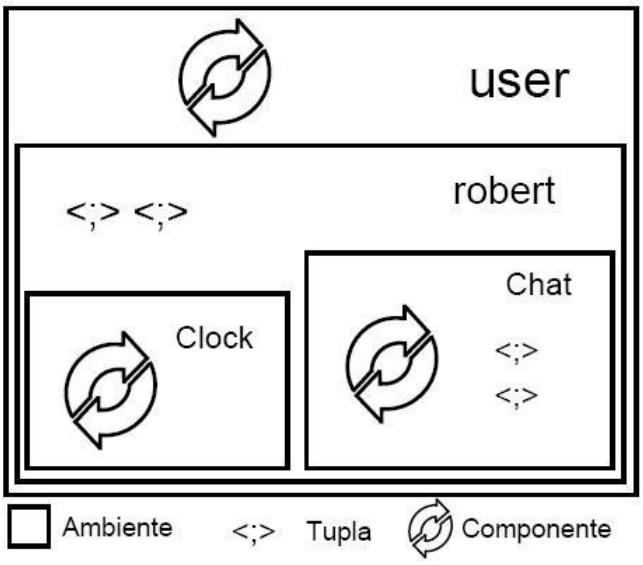

Figura 2.5 - Arquitetura do one.world [26]

Todo o código em one.world é executado em uma máquina virtual, mais precisamente a máquina virtual Java (JVM). Um dos motivos do uso desta máquina virtual é a heterogeneidade inerente a um ambiente pervasivo. Em um ambiente como este os desenvolvedores não tem como prever todo o tipo de dispositivo a aplicação irá executar. Utilizando uma máquina virtual, uma camada de abstração é inserida de modo a fazer com que as aplicações consigam executar em qualquer dispositivo que possua uma máquina virtual. O espaço de tuplas, como dito anteriormente, fornece um modelo de dados homogêneo que simplifica o compartilhamento de dados. Toda a comunicação em one.world, seja ela local ou remota, é feita através de eventos assíncronos. Estes eventos

servem para notificar as aplicações de maneira explícita a respeito das mudanças em seu contexto. Ambientes são os principais componentes da estrutura de one.world. Podendo ser comparados com processos em sistemas operacionais, ambientes hospedam aplicações em execução isolando umas das outras. Eles também servem como *containers* para dados persistentes oferecendo armazenamento associativo de tuplas de modo que aplicações podem ser agrupadas juntamente com seus dados. Adicionalmente, ambientes ainda podem ser aninhados tornando fácil tanto a composição quanto a extensão de aplicações. Um ambiente externo tem controle completo de seus ambientes internos. Esta construção facilita que usuários/desenvolvedores modifiquem o comportamento de suas aplicações sem ter que mudar a aplicação em si.

Os demais serviços disponibilizados em one.world, como migração de código e descoberta de serviços, são construídos com base nos serviços fundamentais apresentados anteriormente. A migração de código em one.world move ou copia um ambiente para um dispositivo diferente. Esta migração, ao contrário da migração de processos tradicionais, não ocorre de maneira transparente e o estado da aplicação migrada se limita ao ambiente sendo migrado. Durante a migração one.world anula as referências a recursos que estão fora do ambiente. Esta prática, segundo o autor [26] é aceitável, pois as aplicações em one.world são concebidas com mudanças constantes em mente.

A mobilidade de código em one.world é considerada fraca, pois o estado que é movido no processo de migração se limita ao ambiente sendo migrado. Esta decisão evita dependências residuais e requer conectividade entre os dispositivos somente durante a migração.

2.4.3 Mobile Gaia

Desenvolvido pela University of Illinois at Urbaba-Champaign, Mobile Gaia [13] é um *middleware* orientado a serviços que integra recursos de diferentes tipos. Este *middleware* disponibiliza funções para a criação e manutenção de coleções de dispositivos, compartilhamento de recursos, possibilitando assim uma integração de maneira transparente.

Mobile Gaia também disponibiliza um *framework* para o desenvolvimento de aplicações tendo uma coleção subjacente em mente. Este *framework* decompõe a aplicação em componentes menores que podem executar em diferentes membros da coleção. Seu principal objetivo é que aplicações que utilizem esta coleção sejam desenvolvidas através de uma interface comum de programação.

Entre os serviços disponibilizados por Mobile Gaia estão os serviços de descoberta de dispositivos, manutenção da coleção de dispositivos, compartilhamento de recursos, bem como um serviço de comunicação entre os dispositivos que fazem parte da coleção. Esta coleção também é chamada de *active space*.

Os serviços em Mobile Gaia são decompostos em componentes possibilitando que somente os componentes necessários a determinado serviço sejam carregados no *middleware* reduzindo assim o consumo de memória e bateria dos dispositivos envolvidos. A fim de obter esta estrutura de componentes, unidades independentes de serviços são identificadas e agrupadas de modo a se obter o serviço desejado. Um serviço de localização, por exemplo, é composto do serviço de sensores, serviço de modelo espacial e de adaptadores de localização.

Os serviços em Mobile Gaia são apresentados em dois papéis: cliente e coordenador. Quando um dispositivo é coordenador de uma coleção, seus serviços possuem responsabilidades adicionais, como integrar serviços similares em dispositivos que fazem parte da coleção, gerenciar os serviços em todos os dispositivos entre outros. O serviço de localização em um dispositivo coordenador, por exemplo, adquire informações sensoriais de todos os dispositivos de sua coleção.

Quando um dispositivo é coordenador de uma coleção, os componentes necessários para a implementação de um determinado serviço são carregados de maneira dinâmica para o *middleware*. De maneira similar, caso o componente coordenador mude seu papel para cliente, os componentes previamente carregados no *middleware* são descarregados e os componentes do novo coordenador são carregados. Esta carga e descarga automática de serviços é feita pelo *framework* de implantação de serviços. Quando um dispositivo muda de papel (de coordenador para cliente por exemplo), o gerenciador de coleções informa ao

framework de implantação de serviços esta mudança e este toma as medidas necessárias, descarregando do *middleware* os serviços do coordenador antigo e carregando os serviços requeridos pelo novo coordenador. A Figura 2.6 mostra a arquitetura do *Mobile Gaia*.

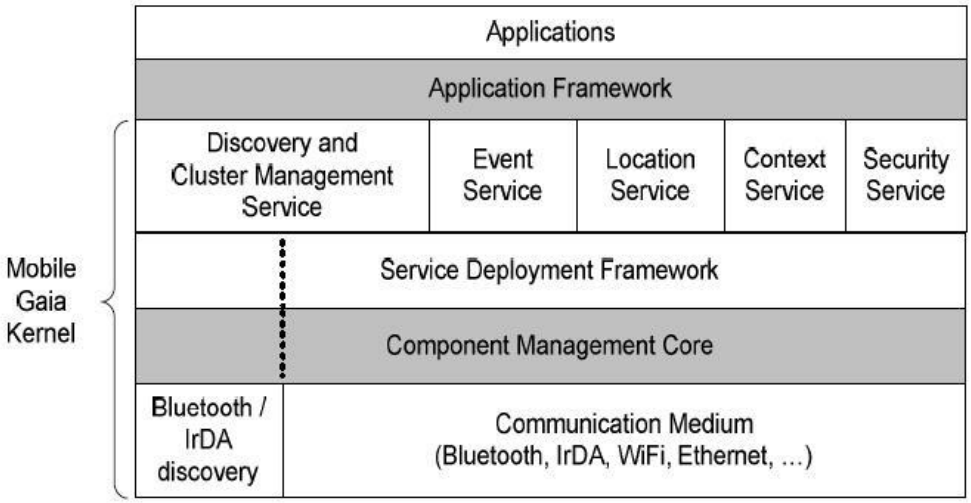

Figura 2.6 - Arquitetura do *Mobile Gaia* [13]

A mobilidade de código em *Mobile Gaia* acontece em nível de componentes. São os componentes básicos de serviço que são movidos entre dispositivos, definindo assim a granulosidade da mobilidade. Enquadrando a mobilidade disponibilizada por *Mobile Gaia* no *framework* apresentado na Seção 2.1, tem-se que a mobilidade é fraca pois o estado de execução dos componentes não é mantido quando estes são carregados no *middleware*. É o *framework* de implantação de serviços quem envia o serviço (código) necessário para os dispositivos, realizando assim uma abordagem de *code shipping*.

2.4.4 ISAM

ISAM [47] é uma arquitetura de software concebida para lidar com a implementação e execução de aplicações pervasivas. Desenvolvido por pesquisadores de universidades do sul do Brasil (UFRGS, UFSM, UFPEL e UNISINOS), tem como um dos objetivos principais prover uma adaptação colaborativa entre a aplicação (programa) e o ambiente de execução (*middleware*).

A arquitetura proposta é organizada em camadas com níveis diferenciados de abstração buscando a manutenção da qualidade dos serviços oferecidos ao usuário móvel através do conceito de adaptação. Uma visão geral desta arquitetura pode ser vista na Figura 2.7.

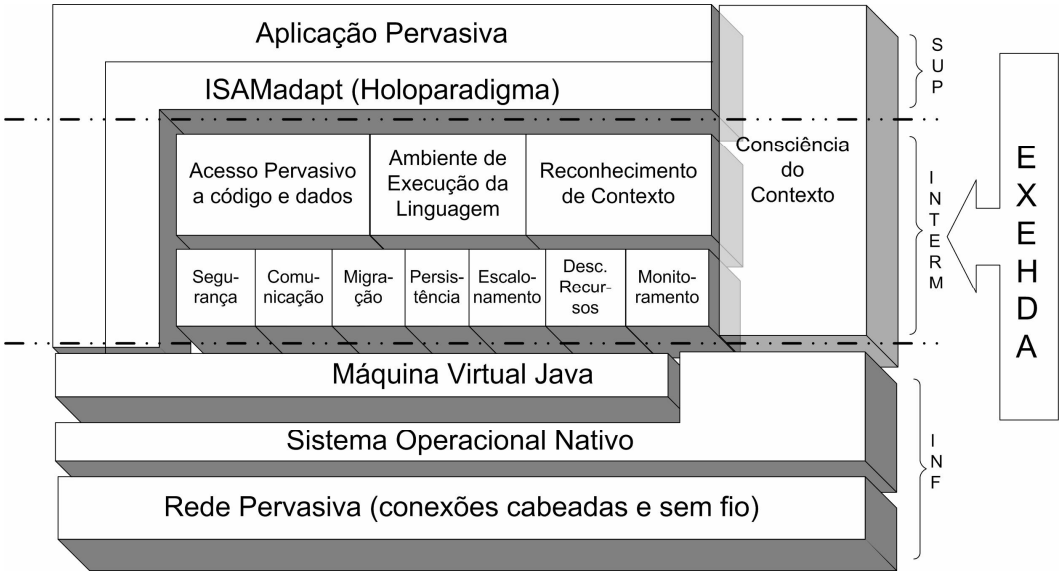

Figura 2.7 - Arquitetura do ISAM [47]

A camada superior da arquitetura é composta pela aplicação móvel distribuída. A construção desta aplicação é baseada em abstrações do Holoparadigma, as quais permitem expressar mobilidade, acrescidas de novas abstrações para expressar adaptabilidade providas pelo ISAMAdapt [2]. A camada inferior, por sua vez, é composta por tecnologias utilizadas em sistemas distribuídos existentes, como sistemas operacionais e a máquina virtual Java, por exemplo.

Na camada intermediária encontram-se três níveis de abstração que formam o núcleo funcional da arquitetura ISAM. O primeiro nível é composto por dois módulos de serviço à aplicação: o escalonamento que é considerado um componente chave da adaptação na arquitetura e o ambiente virtual do usuário, que é composto pelos elementos que integram a interface de interação do usuário móvel com o sistema. É este módulo que permite que uma aplicação sendo executada em uma localização possa ser instanciada e continuar sua execução em outro local sem interrupção suportando desta maneira o desenvolvimento de aplicações que seguem a semântica siga-me.

O segundo nível é responsável pelo contexto da aplicação que é determinado através de informações de quem, quando, o que está sendo realizado e com o que está sendo realizado. A tarefa de obter estes dados é do módulo de monitoramento que atua tanto na parte móvel quanto na parte fixa da rede.

No terceiro nível da camada intermediária estão os serviços básicos do ambiente de execução ISAM que provê à funcionalidade necessária para o segundo nível, cobrindo diversos aspectos como migração de código, replicação otimista e localização. Neste sentido, é nesta camada que a mobilidade de código é tratada dentro da arquitetura de ISAM.

2.5 Conclusão

Atualmente, dentro do contexto da computação distribuída, a mobilidade vem se afirmando cada vez mais como uma forte tendência. Dentre os principais fatores que contribuem para este fato estão o grande poder computacional de dispositivos cada vez menores juntamente com novas tecnologias que emergem visando aproveitar este novo cenário.

Neste capítulo o conceito de mobilidade de código foi analisado do ponto de vista do desenvolvimento de software. Este conceito mesmo não sendo recente, tem atraído à atenção da comunidade científica. Inicialmente um *framework* foi descrito juntamente com uma taxonomia onde as questões que envolvem a mobilidade de código foram expostas. Na seção seguinte os paradigmas de desenvolvimento de software voltados à mobilidade de código foram apresentados juntamente com uma breve discussão envolvendo a utilização dos mesmos no desenvolvimento de aplicações.

No final do capítulo foram apresentados sistemas que utilizam algum tipo de mobilidade de código. Entre estes sistemas destacam-se duas linguagens/plataformas de programação bastante utilizadas atualmente que são Java e .NET.

A idéia deste capítulo foi introduzir de maneira sucinta o conceito de mobilidade de código enfatizando as questões relevantes dentro do contexto do desenvolvimento de software. No próximo capítulo será apresentado o Holoparadigma, a HoloVM que é a

máquina virtual onde os programas que utilizam *byte code* Holo executam e também o ambiente de execução distribuído. O restante deste trabalho está focado na introdução do suporte à mobilidade de código na HoloVM.

3. Holoparadigma

Neste capítulo serão apresentados os conceitos do Holoparadigma, um modelo multiparadigma que estimula a exploração automática de paralelismo e distribuição juntamente com uma descrição detalhada da HoloVM. A HoloVM é uma máquina virtual desenvolvida com o propósito de oferecer suporte à execução de programas que exploram os conceitos oferecidos pelo Holoparadigma. O Holoparadigma será apresentado dentro do contexto do projeto PHolo (*Pervasive Holoparadigm*). Este projeto foi concebido com o objetivo de fornecer suporte à computação ubíqua no Holoparadigma.

Com o intuito de agregar a funcionalidade de mobilidade de código, mais precisamente a mobilidade forte, modificações serão feitas na HoloVM de modo que a mobilidade possa ser explorada de maneira simples do ponto de vista do desenvolvedor de software. Na próxima seção uma descrição breve do Holoparadigma será apresentada com o objetivo de contextualizar o leitor. Informações mais detalhadas deste paradigma podem ser encontradas nas referências bibliográficas.

3.1 Conceitos Básicos

O Holoparadigma [4] é um modelo multiparadigma que integra conceitos de paradigmas básicos (imperativo, orientado a objetos, funcional e lógico) orientado ao desenvolvimento de software paralelo e distribuído. Ele propõe a utilização de processamento simbólico como principal instrumento para o tratamento de informações. Esta característica foi herdada do paradigma de programação em lógica. O Holoparadigma estabelece a utilização de duas unidades de modelagem: o ente, que é sua principal abstração, e o símbolo, que é utilizado para descrever os entes e suas relações. Um ente elementar (Figura 3.1a) é organizado em três partes: interface, comportamento e história. Um ente composto (Figura 3.1b) possui a mesma organização de um ente elementar, porém suportando a existência de outros entes em sua composição (entes componentes), assemelhando-se a um grupo. Neste caso, a história atua como um espaço compartilhado vinculado ao grupo. Cada ente possui uma história que fica encapsulada no seu interior. No

caso de entes compostos esta história é compartilhada pelos entes componentes. Neste sentido vários níveis de encapsulamento de histórias podem existir, porém, entes acessam diretamente a história em dois níveis (o seu e o do seu ente pai). Esta composição de entes (presente em entes compostos) pode variar em tempo de execução de acordo com a mobilidade dos entes. A Figura 3.1c mostra um ente composto por três níveis e exemplifica a história encapsulada.

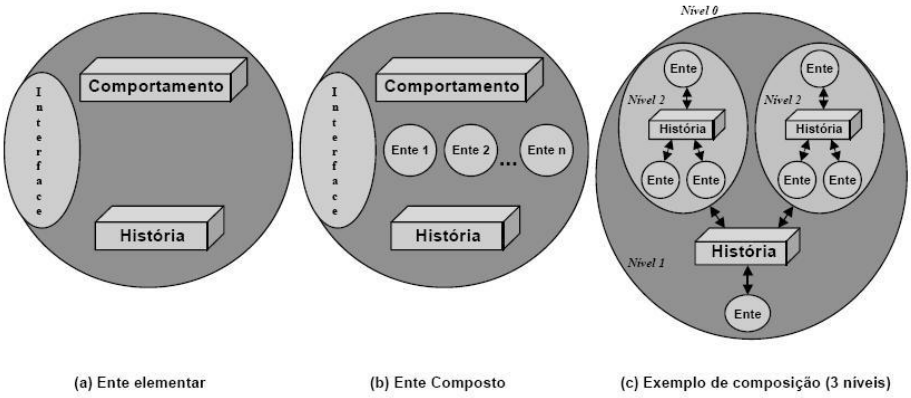

Figura 3.1 - Tipos de Entes

A interface de um ente descreve suas possíveis interações com demais entes. O comportamento contém as ações que definem o comportamento de um ente e a história é um espaço de dados compartilhado no interior de um ente. Esta estrutura, do ponto de vista estrutural, é semelhante a um objeto do paradigma de desenvolvimento orientado a objetos. A principal diferença encontra-se na história, que neste caso atua como uma forma alternativa de comunicação e sincronização.

A mobilidade no Holoparadigma é a característica que permite que entes possam deslocar-se. No âmbito do Holoparadigma existem dois tipos de mobilidade: a mobilidade lógica e a mobilidade de código. A mobilidade lógica relaciona-se com o deslocamento em nível de modelagem, não levando em consideração o ambiente de execução. A mobilidade de código, entretanto, relaciona-se com o deslocamento de entes entre nodos físicos em uma arquitetura distribuída. Desta forma, um ente se move quando se desloca de um nodo para outro. É nesta mobilidade que o restante deste trabalho está focado. A Figura 3.2a

mostra uma possível mobilidade lógica no ente apresentado na Figura 3.1c. A Figura 3.2b mostra uma mobilidade de código que não implica em mobilidade lógica.

No escopo de sistemas distribuídos, um ente pode assumir dois estados de distribuição: centralizado ou distribuído. No estado centralizado um ente encontra-se fisicamente sediado em apenas uma máquina enquanto que no estado distribuído um ente pode agregar entes executando em máquinas diferentes. A Figura 3.2b mostra um possível estado de distribuição para o ente mostrado na Figura 3.1c. Neste caso, a história do ente composto atua como uma memória compartilhada distribuída (DSM – *Distributed Shared Memory*).

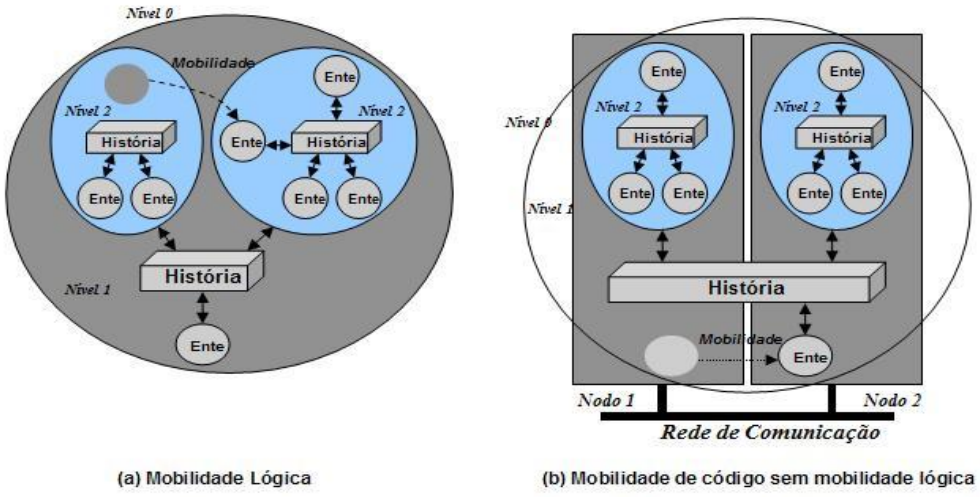

Figura 3.2 - Mobilidade no Holoparadigma

A execução de um programa cria uma estrutura hierárquica de entes, denominada Árvore de Entes (*HoloTree*). Esta árvore implementa o encapsulamento de entes em níveis de composição, conforme proposto pelo Holoparadigma. A *HoloTree* suporta ainda o aspecto dinâmico da política de grupos, mudando continuamente durante a execução de um programa. Ações como a instanciação e a mobilidade de entes são exemplos de ações que modificam a *HoloTree*. A Figura 3.3a exemplifica a *HoloTree* para a organização de níveis mostrada na Figura 3.1c. Um ente composto possui ligações com seus entes componentes, os quais ficam localizados um nível abaixo. Os entes componentes possuem acesso à história e ao comportamento de um ente composto no qual estão inseridos. O ente

composto, por sua vez, possui acesso aos comportamentos dos seus entes componentes. Um ente ainda possui acesso ao comportamento dos demais entes que estão no mesmo nível.

Visando colocar em prática os conceitos estabelecidos pelo Holoparadigma, uma linguagem de programação, a Hololinguagem [5] foi desenvolvida. A Hololinguagem é uma linguagem multiparadigma direcionada para o desenvolvimento de sistemas distribuídos.

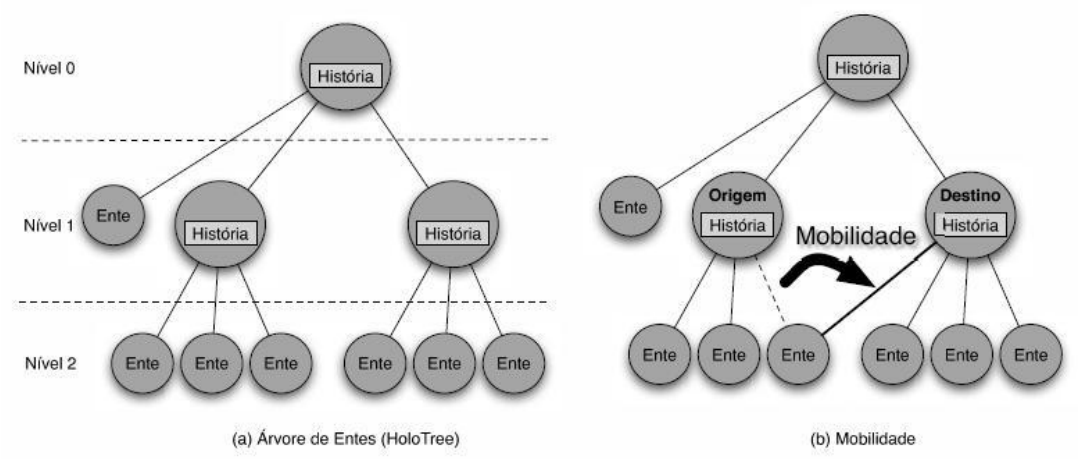

Figura 3.3 - Árvore de Entes (*HoloTree)*

Quando a mobilidade ocorre, torna-se necessária a mudança da visão do grupo dos entes envolvidos. O ente movido possui uma nova visão (novos entes no mesmo nível e um novo ente composto no qual ele está inserido). Caso o ente movido seja composto, a visão de seus componentes não muda. A mobilidade implica uma atualização da composição dos entes origem e destino. Diversas questões conseqüentes a esta mobilidade devem ser tratadas. A mobilidade de um ente elementar, do ponto de vista da *HoloTree* equivale à realocação de uma folha da árvore enquanto que a mobilidade de um ente composto equivale à realocação de um ramo inteiro da árvore contendo diversos entes. A Figura 3.3b mostra a mudança que ocorreria na *HoloTree* para a mobilidade da Figura 3.2a.

Na próxima seção será apresentada a HoloVM, uma máquina virtual desenvolvida com o objetivo de oferecer uma camada de execução a programas baseados no Holoparadigma.

3.2 HoloVM

A HoloVM [23] é uma máquina virtual desenvolvida com o intuito de criar uma camada de abstração entre os programas escritos na Hololinguagem e a máquina física na qual eles executam, possibilitando que um *byte code* Holo seja executado em qualquer plataforma onde exista uma implementação desta máquina virtual. Atualmente existem implementações da HoloVM para as plataformas Windows, Linux, Mac OS e Windows Mobile (*Pocket* PC), compreendendo assim uma diversa gama de dispositivos, desde servidores até PDAs.

A execução de programas pela HoloVM acontece através de um arquivo binário pré-definido. Este arquivo normalmente possui a extensão *hvm* e contém além de uma tabela de símbolos, as instruções (*byte code*) a serem executadas. Qualquer compilador pode gerar, a partir de sua linguagem de origem, um arquivo *hvm* para ser executado pela HoloVM.

A HoloVM realiza processamento simbólico. Sendo assim não existe internamente uma definição explícita de variáveis e tipos. As variáveis são criadas sob demanda e a verificação de tipos é feita durante a execução do *byte code*. Apenas três tipos de dados são considerados para o processamento interno: símbolos, números e *strings*.

A HoloVM fornece um conjunto de instruções específicas que suportam as funcionalidades propostas no Holoparadigma. Entre estas instruções podem ser citadas duas que tem influência sobre a localização dos entes. A primeira, chamada *clone*, é responsável pela criação de um ente na aplicação. A segunda, chamada *move*, faz com que um ente se mova na árvore de execução.

Em sendo uma máquina de pilha, a HoloVM possui duas pilhas. A pilha de operandos e a pilha de controle. A pilha de operandos é utilizada para armazenar os resultados parciais dos cálculos, parâmetros necessários à execução de uma ação ou ente e também informações de retorno destas ações. Dois comandos, *load* e *store* são oferecidos para manipulação desta pilha. A pilha de controle é utilizada somente para controle interno de um fluxo de execução, armazenando informações como o endereço de retorno e variáveis locais a uma ação. Não existem instruções específicas para manipular esta pilha.

Sua manipulação ocorre através de instruções de manipulação de variáveis e de invocação e término de ações. Cada ente possui sua pilha de operandos e de controle, criadas no momento de sua clonagem e destruídas no momento de sua extinção.

A exploração de concorrência na HoloVM é obtida através de multiprogramação leve (*threads*) visando obter vários fluxos de execução dentro do mesmo processo. Internamente a HoloVM, uma unidade básica de execução é chamada de BCE (*Byte Code Executor*). O BCE, por definição, é uma *thread* de execução e sempre está associado a um ente. A História, o Comportamento e a Interface dos entes são implementadados através de *blackboards* [45]. Sendo assim uma invocação feita a eles se resume a uma invocação implícita a um *blackboard*.

3.3 Modelo de Execução Distribuída

Dadas às diversas características que favorecem a exploração da mobilidade de software, mais precisamente a mobilidade de entes, contidas no Holoparadigma, dois modelos de execução distribuída foram desenvolvidos. Estes modelos visam colocar em prática os conceitos estabelecidos no Holoparadigma.

Os modelos desenvolvidos são, respectivamente, o HS (*History Server*) e o HNS (*Holo Naming System*) [8]. Ambos tem o objetivo de permitir a execução de programas que compõe uma modelagem em múltiplas plataformas.

3.3.1 History Server

O HS é basicamente um servidor que disponibiliza um espaço de tuplas que pode ser compartilhado entre diversas HoloVMs. Este servidor implementa ainda o suporte a algumas características que o Holoparadigma possui, dentre as quais se destacam as questões de restrição ao acesso à história. Estas restrições são criadas pelo encapsulamento de entes, e consistem no fato de que um ente pode acessar apenas a sua história e a história de seu ente pai. Ao executar uma aplicação utilizando este sistema, todo acesso à história é mapeado diretamente para o servidor. Quando um ente acessa a história de seu pai, ou a dele próprio, a HoloVM se encarrega de mapear este acesso para uma comunicação com o HS. Este então mantém espaços de memória independentes para cada ente criado no

ambiente de execução distribuído. Desta maneira entes em diferentes máquinas podem se comunicar.

Na Figura 3.4 é mostrada a representação de um ambiente de execução utilizando o HS juntamente com as possíveis interações com uma ou mais HoloVMs. No **passo 1** a **HVM a** informa a criação de um ente para o servidor. Tendo feito isto, para que dois entes possam comunicar-se, basta que eles possuam um ente em comum.

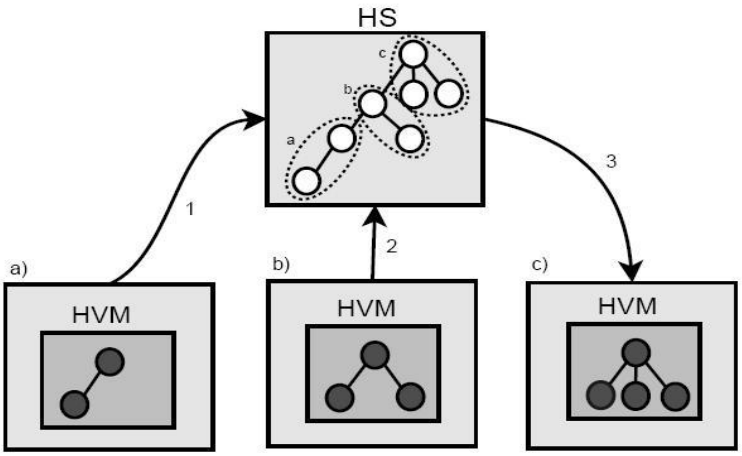

Figura 3.4 - Funcionamento do HS

Quando uma HoloVM é executada com o HS, todo o ente criado gera uma mensagem para o servidor que o registra. Esta é a etapa inicial de um processo que permite que o servidor mantenha a consistência com relação aos programas em execução. A localização da HoloVM que enviou a mensagem ao servidor não é conhecida pelo HS. Somente a HoloVM é quem conhece a localização do HS. Após o registro de um ente no HS, qualquer comando de acesso à história executado por ele é mapeado para o servidor. Da mesma maneira um comando *move* também implica em mudança na estrutura mantida no servidor. Os **passos 2** e **3** exemplificam as **HVMs b** e **c**, interagindo com o servidor para acessar a história de um ente.

A implementação deste modelo permitiu que os conceitos propostos pelo Holoparadigma fossem aplicados a um ambiente distribuído. Entretanto o HS apresenta algumas limitações. A primeira delas é que ele pode ser visto como um ponto central de falhas devido sua característica centralizada. Outra limitação é que este modelo

implementa apenas abstrações para acessos à história dos entes. No entanto esta não é a única abstração do Holoparadigma a ser explorada em um ambiente distribuído. Desta maneira um novo modelo que atende melhor aos requisitos do Holoparadigma foi definido. Este modelo é apresentado na seção seguinte.

3.3.2 Holo Naming System

O principal objetivo do HNS [43] é prover um serviço que possibilite a execução de programas Holo distribuídos entre diversas HoloVMs. Dois problemas principais devem ser resolvidos de modo a fornecer este suporte. O primeiro refere-se à localização de entes em um ambiente distribuído. O segundo implica o desenvolvimento de uma camada de execução que torne possível que duas HoloVMs possam dar suporte ao uso de todos os recursos propostos no Holoparadigma que envolvam interação entre entes.

A resolução do primeiro problema ocorreu com o desenvolvimento de uma solução baseada no DNS [31]. Desta maneira o HNS contém uma lista de entes de um determinado ambiente juntamente com os endereços IP dos dispositivos onde eles estão executando.

Com a criação do HNS, cada *HoloTree* existente em cada HoloVM tem apenas uma visão parcial do cenário. Apenas o servidor é quem tem a visão total pois todas as HoloVMs envolvidas reportam ao servidor qualquer alteração ocorrida em sua estrutura. Esta estrutura mantida pelo HNS também é chamada de *Distributed Holo Tree* (*DHoloTree*).

As mensagens trocadas entre o servidor e as HoloVMs podem ser separadas em dois grupos. No primeiro grupo estão as que servem para informar o servidor sobre uma alteração na árvore juntamente com as mensagens que requisitam ao servidor informações a respeito da localização dos entes. As instruções *clone* e *move* sempre que executadas irão gerar uma mensagem para o HNS, pois ambas alteram a estrutura interna da HoloVM que as executou. Desta maneira o HNS sempre mantém uma visão atualizada da árvore distribuída.

Uma camada de software foi desenvolvida para executar a comunicação entre a HoloVM e o HNS. Através dela a HoloVM pode informar ao HNS a respeito de

modificações em seu estado interno bem como executar consultas. Um protocolo de camada de aplicação foi desenvolvido para a troca de mensagens. Este protocolo caracteriza-se por ser bastante simples e reduzido, de modo a gerar um pequeno *overhead* de comunicação.

No segundo grupo encontram-se as instruções de acesso a história, comportamento e interface juntamente com instruções que o programador pode utilizar para perguntar quem é o pai ou mesmo quem são os entes que compartilham o mesmo pai.

Quando determinado ente necessita interagir com outro, e este último não se encontra na mesma HoloVM, é necessário então executar uma consulta no HNS para descobrir a localização deste ente. O servidor, por sua vez, retorna um URI (*Uniform Resource Identifier*) para a HoloVM. De posse desta localização, as HoloVMs podem então interagir diretamente. Neste estágio elas podem realizar acessos remotos tais como:

- Acessar a história de determinado ente localizado em uma HoloVM remota;
- Disparar ações pertencentes ao comportamento de entes remotos;
- Acessar ações que estão publicadas na interface dos entes remotos.

Durante a execução de uma HoloVM utilizando o HNS, existem dois momentos distintos. No primeiro, como demonstrado na Figura 3.5, acontece a descoberta do serviço pela HoloVM, onde a **HVM a** descobre um HNS disponível (**passo 1**) e este retorna uma mensagem para a HoloVM informando sua localização (**passo 2**). Depois disso a HoloVM utiliza o suporte oferecido pelo HNS, sincronizando as informações de sua HoloTree com a DHoloTree do servidor (**passo 3**). Depois de sincronizados os contextos, a HoloVM executa utilizando o suporte oferecido pelo HNS. Ainda na Figura 3.5 dois passos que permitem a comunicação entre dois entes são mostrados. No passo 4 a **HVM c** faz uma consulta ao servidor para descobrir a localização da HoloVM que possui o ente com o qual ela quer ser comunicar. Caso o servidor contenha esta informação ela é retornada para a **HVM c** (**passo 5**). Com base nesta informação ela envia a requisição da ação desejada para a **HVM b** (**passo 6**) que irá executar a ação e retornar o resultado (**passo 7**). Este processo

ocorre de forma transparente para o usuário, facilitando o desenvolvimento de programas distribuídos.

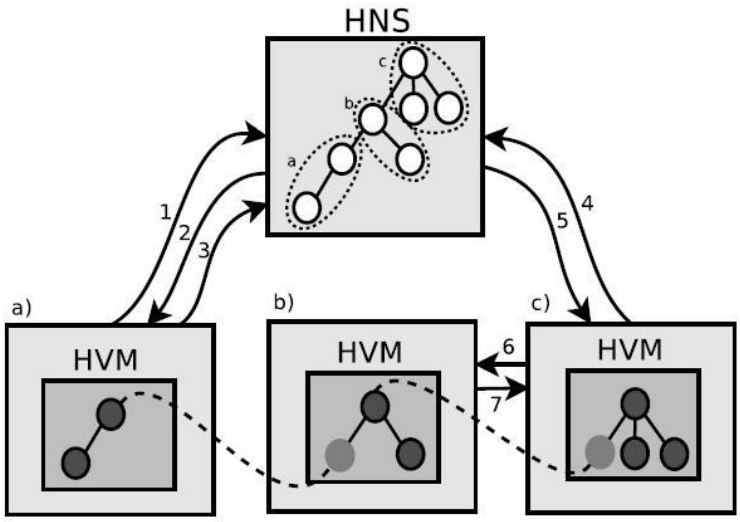

Figura 3.5 - Representação do funcionamento do HNS

O HNS é composto por três módulos independentes. O primeiro é o servidor HNS que provê a estrutura para o armazenamento da *HoloTree* juntamente com a implementação do protocolo de comunicação para interagir com as HoloVMs. O segundo módulo implementa uma camada de comunicação entre a HoloVM e o servidor HNS. Este módulo disponibiliza métodos para anunciar a criação de entes, suas ações de mobilidade, bem como fazer consultas a respeito da estrutura da *DHoloTree* juntamente com o endereço dos entes que a compõem. O terceiro módulo é responsável pela comunicação entre HoloVMs e consiste de duas partes. Uma delas é um servidor acoplado na HoloVM que fica aguardando por requisições. Este servidor tem acesso a todas as estruturas de dados dos entes em execução na HoloVM e é capaz de executar consultas locais e retornar o resultado para HoloVMs remotas. A outra parte é responsável por fazer requisições sempre que um ente precise interagir com algum ente disponível em outra HoloVM.

3.4 Conclusão

Este capítulo apresentou o Holoparadigma juntamente com os modelos propostos para dar suporte à execução distribuída de programas que o utilizam. Através de abstrações

simplificadas para a representação de ambientes onde aplicações móveis executam, ele apresenta uma proposta inovadora para o desenvolvimento de aplicações distribuídas.

Também apresentado neste capítulo uma máquina virtual chamada HoloVM. É ela quem oferece suporte à execução de programas baseados em *byte code* Holo. A HoloVM oferece suporte nativo a todas as características do paradigma permitindo que serviços sejam implementados diretamente no ambiente que suporta a execução. Esta característica permite que programas possam ser mapeados diretamente para instruções da VM.

Duas soluções foram propostas para oferecer suporte à execução distribuída. Inicialmente foi implementado o HS para uma validação preliminar. Dadas às diversas limitações do HS, o HNS foi proposto de modo a suprir as limitações do HS. Através do HNS programas Holo podem empregar de maneira distribuída diversos conceitos expostos pelo Holoparadigma.

Entretanto, até então o Holoparadigma suportava apenas a mobilidade lógica, sendo a mobilidade de código presente apenas na especificação. Com o objetivo de preencher esta lacuna, o próximo capítulo apresenta o modelo de mobilidade forte de código desenvolvido.

4. Holo*Go* – Mobilidade de Código na HoloVM

Este capítulo apresenta o modelo proposto para mobilidade forte de código. O objetivo deste modelo é disponibilizar esta funcionalidade entre HoloVMs. Este modelo pode ser visto como uma camada de software acoplada a HoloVM de modo a oferecer a mobilidade forte de código sem a necessidade de modificação na Hololinguagem. Antes de Holo*Go* a HoloVM suportava apenas a mobilidade lógica de entes.

4.1 Modelo Proposto

Sendo utilizada como plataforma de execução para programas baseados em *byte code* Holo, a HoloVM, em uma analogia com o *framework* apresentado no Capítulo 2, pode ser vista como um ambiente computacional. Neste sentido, o ente é quem irá se mover entre HoloVMs e pode ser visto como uma unidade de execução. A Figura 4.1 mostra a HoloVM inserida no contexto do *framework* apresentado.

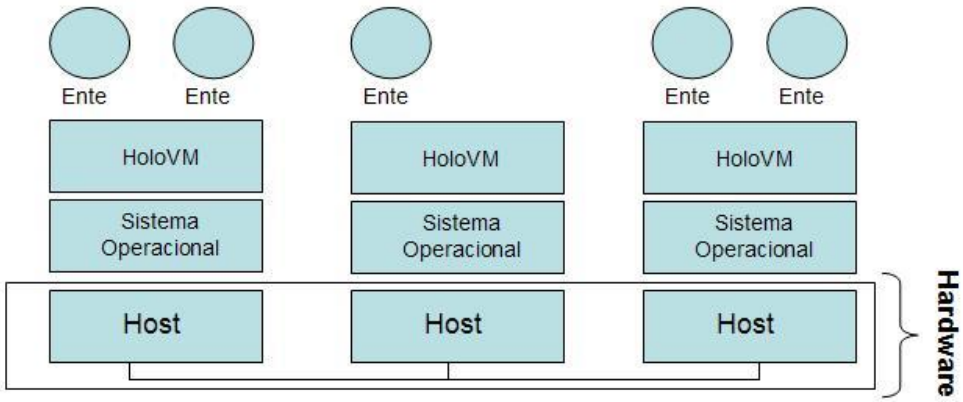

Figura 4.1 - Sistema de Código Móvel tendo a HoloVM como ambiente computacional

O objetivo do modelo é disponibilizar na HoloVM a mobilidade forte de código, ou seja, o ente além de se mover, move consigo o seu estado de execução que é composto por duas pilhas (operandos e controle), interface, comportamento e história. A estrutura interna de um ente é mostrada na Figura 4.2.

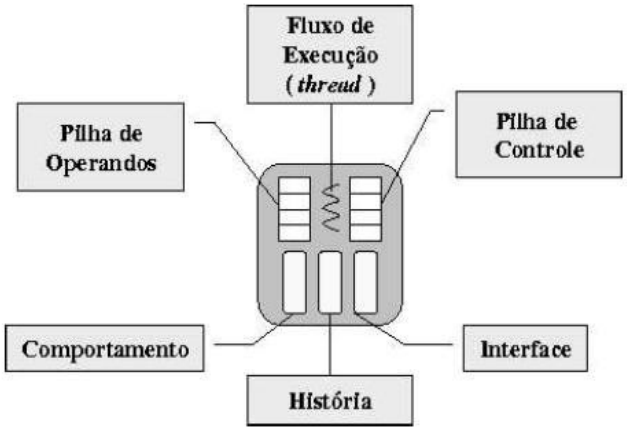

Figura 4.2 - Estrutura interna de um ente.

Os seguintes passos [32] são necessários para obtenção da mobilidade forte de código entre entes:

- Parar a execução de um ente;
- Serializar os dados de um ente;
- Transmitir os dados de um ente para outro ambiente;
- Reconstruir, a partir dos dados recebidos, o ente e seu estado de execução no ambiente destino.

Toda a funcionalidade referente aos passos necessários é disponibilizada por Holo*Go*. Um ente ao necessitar de uma mobilidade de código utiliza as funcionalidades disponibilizadas por Holo*Go*. Neste sentido é ele quem efetiva os passos necessários para a execução de uma mobilidade de código. Ao ser recebido por outra HoloVM, é o Holo*Go* da HoloVM de destino quem irá, a partir dos dados serializados recebidos, recriar o ente e reiniciar sua execução inserindo-o na *HoloTree*. Na versão atual de Holo*Go*, caso o ente sendo movido esteja aguardando pelo resultado de uma ação, este invariavelmente espera por este retorno.

O suporte oferecido pelo HNS é uma peça essencial deste modelo, pois é através das funcionalidades dele que os entes podem obter informações de localização a respeito de outros entes. Ao executar uma mobilidade, através do comando *move*, o ente estará, de maneira implícita, perguntando ao HNS qual a localização de seu ente destino. Ao receber esta informação, as medidas necessárias para executar a mobilidade podem ser tomadas. Neste sentido a mobilidade de código acontece de maneira implícita, ou seja, o ente executando o comando *move* não sabe de antemão se ele necessitará ser movido para outra HoloVM. Esta abordagem reduz o esforço de desenvolvimento de uma aplicação, no sentido que a mobilidade forte é explicitada por apenas uma instrução.

A Figura 4.3 mostra a comunicação realizada por Holo*Go* entre as HoloVMs e o HNS no momento em que um ente executa uma mobilidade. No passo um, ao executar a mobilidade, a HoloVM pergunta ao HNS onde está localizado o ente destino. No passo dois a resposta é obtida. No passo três a HoloVM comunica-se diretamente com a HoloVM de destino e a mobilidade de código é efetivamente executada. No passo quatro a HoloVM de destino informa ao HNS a localização no novo ente e no passo cinco o HNS atualiza sua estrutura.

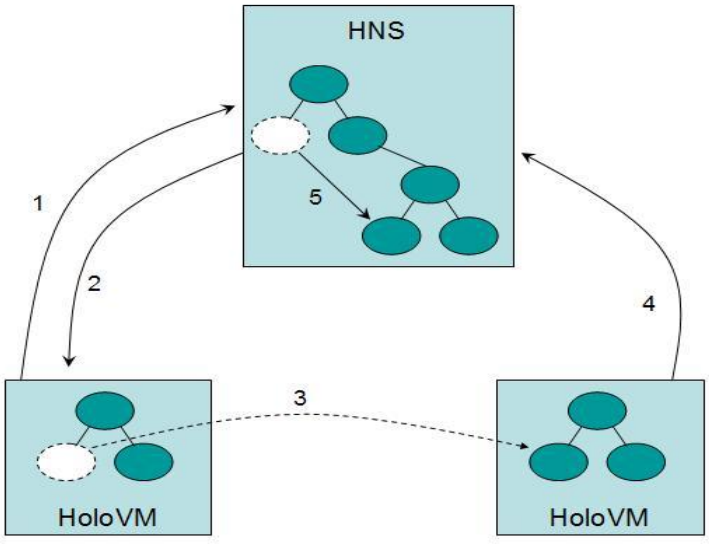

Figura 4.3 - Passos realizados no momento de uma mobilidade de código.

A mobilidade forte de código em Holo*Go* pode ser pró-ativa ou reativa. Um ente pode executar uma ação em outro ente que pode vir a resultar em uma mobilidade de código deste ente onde a ação foi executada, caracterizando assim uma mobilidade de código reativa. Da mesma maneira, o próprio ente pode mover-se para outro ambiente caracterizando uma mobilidade de código pró-ativa.

Os recursos utilizados por um ente no momento de sua migração podem ser tratados de acordo com quatro técnicas [21]: remover o vínculo e referenciar o recurso através da rede, vincular novamente o recurso no ambiente de destino, copiar o recurso juntamente com o ente ou ainda mover o recurso junto com o ente. Dependendo do tipo de recurso sendo utilizado uma opção pode ser mais vantajosa em detrimento de outra. Certos tipos de recursos podem ser descartados, sabendo-se de antemão que estarão disponíveis no ambiente computacional (HoloVM) de destino. Outros, como por exemplo um arquivo não existente no ambiente de destino, devem ser tratados de outra maneira.

Uma questão chave a ser tratada quando da ocorrência de uma mobilidade forte de código refere-se ao tratamento da *Constant Pool*. Haja visto que ela é carregada no momento em que o arquivo contendo a descrição de um programa Holo é lido do disco, e que a posição de cada símbolo armazenado nela está diretamente relacionada com o *byte code* executado, sua modificação em tempo de execução torna-se dificultada. Desta maneira esta questão foi abordada adotando-se a política de ao mover um ente, fazer com que ele leve junto sua *Constant Pool*. Sendo assim, ao ingressar na HoloVM de destino e reiniciar sua execução, todas as referências feitas à *Constant Pool* são feitas a sua *Constant Pool* local e não a *Constant Pool* da HoloVM destino. Através desta abordagem, toda vez que um ente é movido fisicamente, ele leva junto sua *Constant Pool* e passa a referenciá-la toda vez que uma instrução que a acesse seja executada. Esta situação é mostrada na Figura 4.4.

Um ente ainda pode estar compartilhando recursos com mais entes ao mover-se para outro ambiente. Neste caso, se o ente leva o recurso consigo, os demais entes terão que acessar este recurso por meio de referências à rede, ou, caso o recurso não saia do ambiente de origem, o ente movido deve acessá-lo através da rede. Em ambientes onde a conexão de rede é limitada, como um ambiente de dispositivos móveis conectados através de conexões

sem fio, esta abordagem pode ocasionar falhas de execução. Neste contexto um recurso pode ser considerado fixo ou móvel. Um recurso fixo está permanentemente ligado a uma HoloVM, enquanto que um recurso móvel pode ser movido ou copiado para outras HoloVMs. Neste trabalho, o único recurso tratado em uma mobilidade de código é a *Constant Pool* que é movida juntamente com o ente sendo movido.

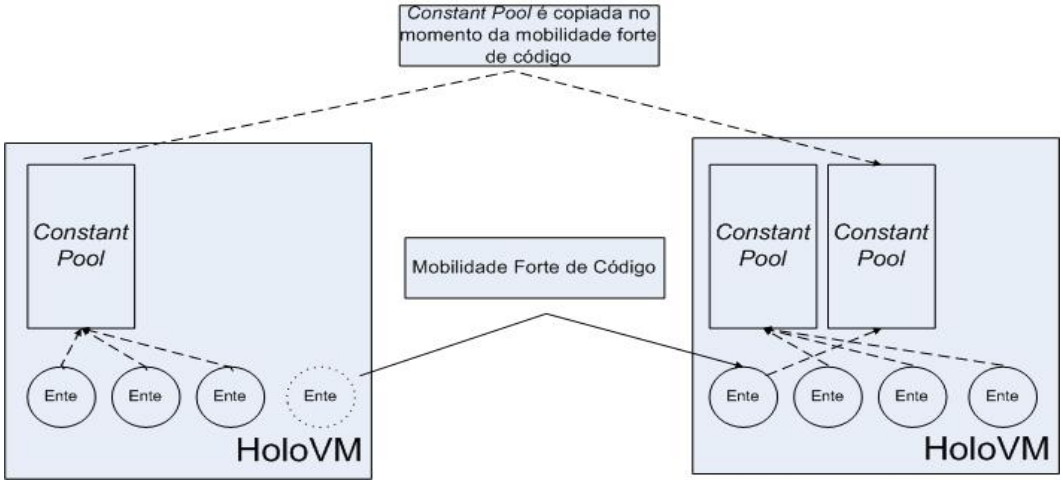

Figura 4.4 - Cópia da *Constant Pool* em uma mobilidade de código

Uma possível abordagem para facilitar o gerenciamento de recursos seria fazer com que a HoloVM, além de exportar ao HNS a localização dos entes que nela executam, exporte também informações sobre os recursos disponíveis. Desta maneira, um ente poderia consultar através do HNS sobre demais ambientes que possuam os recursos necessários para sua execução. Através destas informações um ente poderia escolher ambientes computacionais baseado nos recursos disponíveis por eles.

Em Holo*Go*, um ente composto ao ser movido leva consigo seus entes componentes. Ao detectar uma mobilidade de código, o ente composto deve avisar seus entes componentes a respeito para que estes parem suas execuções e estejam aptos a serem movidos. As ações que um ente tenha invocado e ainda não tenha recebido resposta devem ser aguardadas antes de uma mobilidade de código. Desta maneira a mobilidade não acontece de maneira imediata nestes casos.

Do ponto de vista da *HoloTree*, ao mover um ente composto, todo um ramo da árvore é movido enquanto que a mobilidade de um ente elementar implica na

movimentação de uma folha da árvore. A mobilidade de código de um ente composto é mostrada na Figura 4.5. Os mesmos passos (denotados pelos números nas setas) realizados na mobilidade de código mostrada na Figura 4.3 são realizados. A diferença é que neste caso um ramo inteiro da árvore é movido ao invés de apenas uma folha.

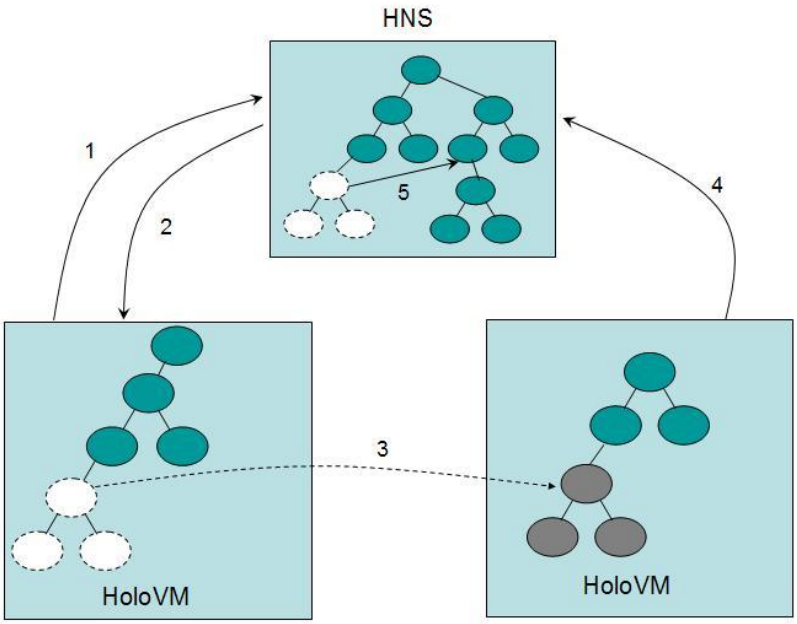

Figura 4.5 - Mobilidade de código em um ente composto

4.2 Modificações Realizadas na HoloVM

A fim de oferecer o suporte à mobilidade forte de código, diversas funções internas à HoloVM tiveram que ser modificadas. Para cada classe utilizada na execução de um programa baseado em *byte code* Holo, funções de serialização e de deserialização foram adicionadas. As classes envolvidas diretamente na execução de um ente são demonstradas na Figura 4.6.

A classe *BytecodeExecutor* corresponde a uma *thread* de execução. É nesta classe que estão contidas as funções referentes a cada um dos *byte code* definidos na HoloVM. As pilhas de operando e de execução estão contidas e são manipuladas por esta classe. Ela também possui como membro uma instância da classe *ConcurrentTask* onde estão contidos o comportamento e a história de um ente. No diagrama também é mostrada a classe que

implementa a *ConstantPool*. É ela quem mantém uma lista contendo todas as constantes necessárias para a execução de um ente, atuando como uma tabela de símbolos. Tanto a classe *Behavior* quanto a classe *History* utilizam o conceito de *blackboards* a fim de armazenar dados. Na classe *Behavior* ficam armazenadas as ações que um ente pode executar, enquanto que a classe *History* é utilizada para a comunicação e também sincronia entre entes.

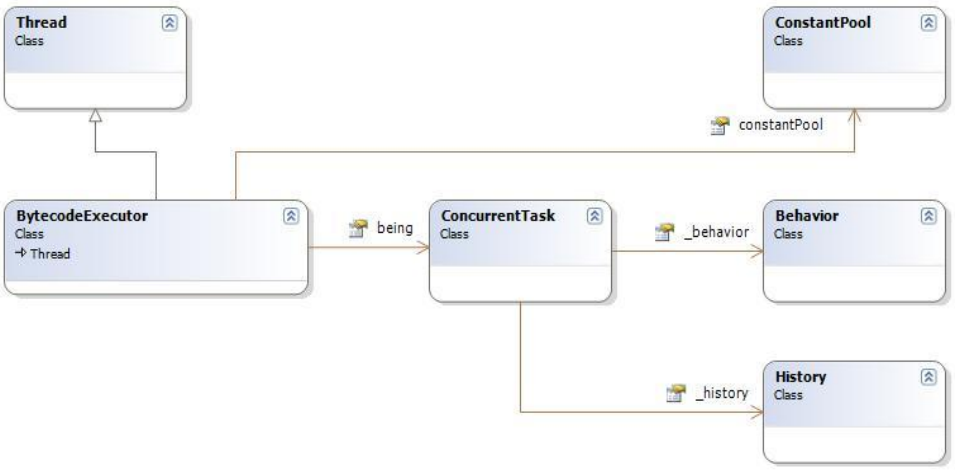

Figura 4.6 - Diagrama descrevendo as principais classes envolvidas

Ao ser realizada uma mobilidade de código, é o conteúdo destas classes que deve ser salvo e enviado ao ambiente computacional de destino.

Os módulos referentes à comunicação entre HoloVMs também foram modificados. Toda a estrutura legada foi aproveitada. Sendo assim, utilizou-se o mesmo padrão de envio e recebimento de mensagens entre HoloVMs. O formato de uma mensagem que contém um ente é mostrado na Figura 4.7.

Figura 4.7 - Formato da mensagem utilizada para enviar um ente

O campo **Ente** identifica ao ente sendo movido. O campo **Pai** identifica ao ente no qual o ente movido será inserido. O campo **Tipo** especifica o tipo da mensagem sendo enviada. Ele é utilizado na decodificação da mensagem pelo módulo servidor da HoloVM de destino. O último campo da mensagem corresponde aos dados serializados que compreendem o ente sendo enviado. No caso de um ente composto, seus entes filho são enviados também através deste campo.

O código original da instrução *move* foi modificado de modo a prover a mobilidade forte de código. A Figura 4.8 mostra o código da instrução em sua versão original. Exceções e verificações de erro foram eliminadas de modo a favorecer a legibilidade do código fonte.

```
1   string source  = pop().valueToStr();
2   string target  = pop().valueToStr();
3
4   if((*_dynamicBeings)[target]== NULL){
5         remote_father = true;
6         HVMClient c((*_dynamicBeings)[source],host);
7         c.move(target);
8   }
9   else{
10        remote_father = false;
11        (*dynamicBeings)[source]->out()->dec_beings_in();
12        (*dynamicBeings)[target]->inc_beings_in();
13        (*dynamicBeings)[source]->out(*dynamicBeings[target]);
14  }
```

Figura 4.8 - Código original da instrução *move*

Ao ser executada, a instrução *move* espera que dois argumentos estejam contidos na pilha de operandos: o ente a ser movido (*source*) (linha 1) e o ente para onde este ente deve ser movido (*target*) (linha 2). Antes da concepção de Holo*Go* apenas a mobilidade lógica era oferecida. Desta forma, quando uma instrução *move* era executada, a HoloVM inicialmente pesquisava pelo local deste ente de destino (*target*) (linha 4). Caso este ente estivesse situado localmente, apenas o ponteiro para o ente pai deste ente sendo movido era modificado de modo a refletir a nova configuração, como é mostrado no código da linha 10 até a linha 13. Caso este ente não estivesse localizado na mesma HoloVM, apenas uma variável (*remote_father*) era modificada de modo a indicar que este ente possui um pai que não estava localizado nesta HoloVM como visto no código da linha 5 a linha 7. Neste caso, qualquer mensagem deste ente para seu pai resultaria em uma mensagem enviada pela rede.

Em ambos os casos (*target* local ou *target* remoto), as modificações realizadas na árvore são comunicadas ao HNS para que este possua uma visão precisa do ambiente de execução distribuído.

As modificações introduzidas no código da HoloVM de modo a oferecer suporte à mobilidade forte de código são mostradas na Figura 4.9. Verificações de condições de erro, bem como tratamento de exceções também foram removidas do código de modo a promover a legibilidade do código fonte.

```
1  string source  = pop().valueToStr();
2  string target  = pop().valueToStr();
3  if((*_dynamicBeings)[target]== NULL){
4        (*dynamicBeings)[source]->WaitForChilds();
5        (*dynamicBeings)[source]->SaveStatus();
6        HVMClient c((*_dynamicBeings)[source],host);
7        Address* addr = c.GetBeingHVmAddress(target);
8        c.SendBeing(addr,source,target);
9        (*dynamicBeings)[source]->StopVM = true;
10 }
11 else{
12       remote_father = false;
13       (*dynamicBeings)[source]->out()->dec_beings_in();
14       (*dynamicBeings)[target]->inc_beings_in();
15       (*dynamicBeings)[source]->out(*dynamicBeings[target]);
16 }
```

Figura 4.9 - Código da instrução *move* modificado

Através do uso de Holo*Go*, ao ser executada uma instrução *move*, a HoloVM também pesquisa pelo local referente ao ente de destino (ente *target*). Caso este esteja situado localmente, apenas a mobilidade lógica é executada como no caso anterior (linhas 12 a 15). Caso contrário a HoloVM notifica os entes filhos deste ente prestes a ser movido para que estes parem sua execução (linha 4). Após o estado de execução deste ente, juntamente com o estado de execução dos entes filhos (caso haja algum) é salvo (linha 5). Neste ponto o módulo cliente da HoloVM é utilizado para enviar o ente serializado para a outra HoloVM. Sendo assim, uma interação com o HNS é feita através da instanciação de um objeto da classe HVMClient que é quem disponibiliza as funções referentes à parte cliente da HoloVM. Este objeto instanciado requisita ao HNS o endereço da HoloVM onde o ente de destino está localizado. De posse desta informação o ente pode ser enviado à HoloVM de destino. Após o envio, a HoloVM seta a variável `StopVM` como verdadeira

no ente que foi enviado de modo que sua execução nesta HoloVM seja interrompida. Desta maneira todas as estruturas alocadas por este ente são retiradas da memória.

O estado de execução de um ente, quando salvo, é empacotado em uma mensagem como a mostrada na Figura 4.7. Ao chegar à HoloVM de destino, quem manipula esta mensagem é o módulo servidor da HoloVM. Implementado através da classe HVMServer, este módulo fica constantemente aguardando por requisições em uma porta específica. Desta maneira, ao receber uma mensagem, inicialmente a mesma é carregada e seu tipo é verificado. Caso seja uma mensagem contendo um ente, esta classe invoca as rotinas necessárias para que o ente seja instanciado novamente e seu estado de execução recuperado. Após isso, este ente é inserido na HoloTree da HoloVM que o recebeu e sua execução é reiniciada do ponto onde ela havia parado anteriormente.

Através desta abordagem, aplicações que utilizam mobilidade de código podem ser modeladas utilizando o Holoparadigma através da utilização dos três paradigmas de desenvolvimento apresentados no Capítulo 2. Um ente pode executar uma ação em seu ente pai e esta ação pode resultar em um ente irmão sendo movido para dentro dele, caracterizando desta maneira o paradigma COD (*Code on Demand*). Outro cenário possível é um ente que envia um ente filho para outra HoloVM a fim deste executar uma tarefa. Neste caso esta mobilidade estaria caracterizando o paradigma REV (*Remote Evaluation*). Um ente ainda pode decidir mover-se por sua própria vontade, caracterizando assim o paradigma MA (*Mobile Agent*).

Outra característica desta abordagem é que ao oferecer a mobilidade forte de código, é provida também a mobilidade fraca. Um ente ao ser movido pode carregar consigo apenas o código de outro ente e instanciá-lo através do comando *clone* em diferentes HoloVMs. Desta maneira, um ente pode viajar pelo ambiente instanciando novos entes em cada ambiente computacional por onde ele passa.

4.3 Conclusão

Este capítulo apresentou detalhadamente o modelo de mobilidade forte de código desenvolvido com o Holoparadigma em mente. Dentre suas principais características encontra-se a facilidade de desenvolvimento de uma aplicação que utilize mobilidade forte

de código, visto que a mesma ocorre de maneira transparente para o desenvolvedor através do comando *move*. Outra característica deste modelo é que ele não impõe nenhuma modificação a Hololinguagem, fazendo com que programas escritos anteriormente continuem funcionando mesmo após sua implementação.

Diversos sistemas de código móvel são baseados na linguagem Java [25]. A maioria destes sistemas, entretanto, disponibiliza apenas mobilidade de código fraca dado que diversas dificuldades devem ser transpostas na implementação de mobilidade forte nesta plataforma [40] [10]. A maioria destas dificuldades refere-se ao tratamento de *threads*. Dado que o código da JVM é mantido por uma empresa privada, soluções para mobilidade forte nesta plataforma implicam em um custo de implementação elevado onde diversas questões devem ser levadas em conta. Neste sentido, a HoloVM sendo uma máquina virtual com código aberto e desenvolvida especialmente para suportar programas baseados no Holoparadigma, facilita o desenvolvimento de uma solução baseada em mobilidade forte de código.

Questões referentes à implementação do modelo serão apresentadas no próximo capítulo.

5. Aspectos de Implementação

Antes de iniciar o processo de prototipação do modelo proposto, um estudo das estruturas internas a HoloVM foi realizado. Através deste estudo, foram identificadas todas as estruturas necessárias a uma mobilidade forte de código. Foi constatado também que a HoloVM não foi projetada de modo a prover esta funcionalidade. Esta característica impactou em dificuldades quando da implementação do modelo.

5.1 Descrição do Protótipo

Holo*Go* foi implementado em ANSI C++ de modo a ser compatível com qualquer plataforma para a qual a HoloVM seja portada. Toda a troca de mensagens entre HoloVMs no que diz respeito à mobilidade de código é realizada através dos módulos de software disponibilizados pelo HNS e pelos módulos cliente e servidor acoplados a cada HoloVM. Nesta abordagem, uma HoloVM assume o papel de servidor, de cliente, ou até mesmo ambos os papéis. Na implementação atual não foram considerados obstáculos constantemente presentes em uma rede aberta, tais como NATs e *firewalls*.

Nenhuma modificação referente à mobilidade de código foi realizada no servidor HNS. Modificações foram realizadas apenas nos módulos servidor e cliente da HoloVM, onde funcionalidades de envio e recebimento de *byte code* Holo foram adicionadas. Nas modificações realizadas nestes módulos toda a estrutura original foi mantida. O protocolo utilizado na comunicação entre HoloVMs foi estendido de modo a suportar o envio e recebimento de entes. Neste sentido apenas as classes envolvidas diretamente nesta comunicação tiveram que ser modificadas. Os passos realizados pelo protótipo no momento de uma mobilidade de código são mostrados no diagrama de seqüência da Figura 5.1.

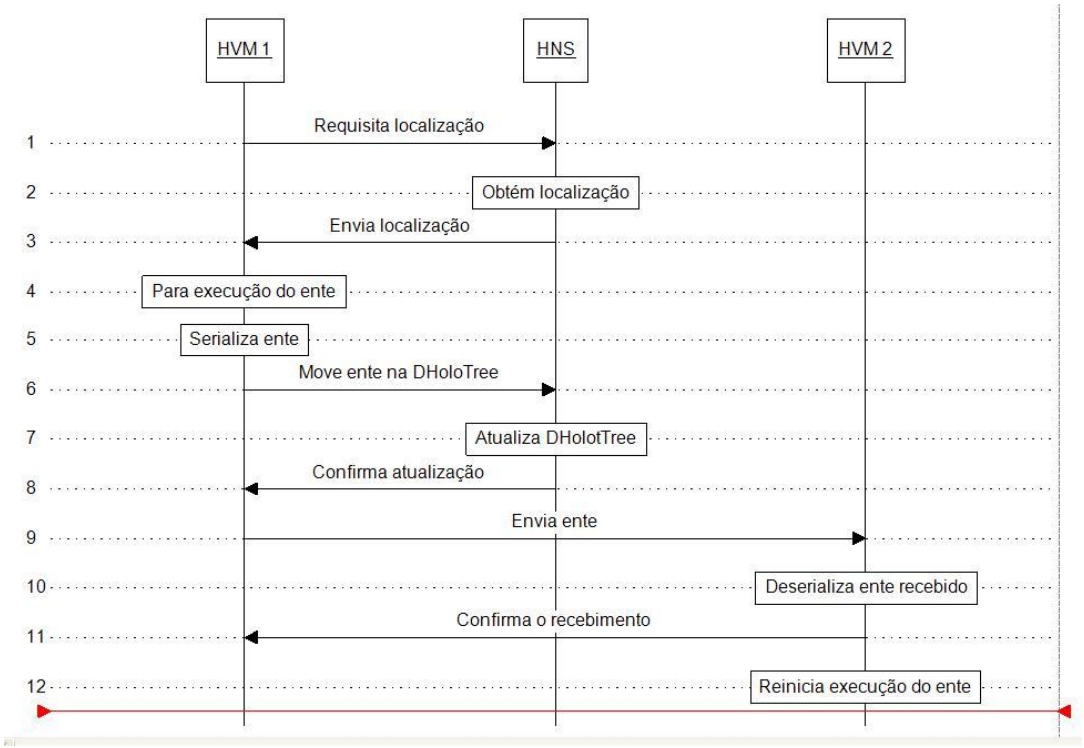

Figura 5.1 - Diagrama de seqüência de uma mobilidade de código

As funções de serialização e deserialização das classes envolvidas na execução de *byte code* Holo foram implementadas com a utilização da biblioteca Boost [29]. A Boost é uma biblioteca de classes desenvolvida em ANSI C++. Através de uma semântica simples, a adição destas funcionalidades, tanto em classes simples quanto em classes complexas tornou-se bastante facilitada, reduzindo desta maneira a complexidade requerida no processo de desenvolvimento. Através do uso desta biblioteca o formato das mensagens enviadas em uma mobilidade de código torna-se bastante flexível. No protótipo atual elas são transmitidas em formato texto, mas com modificações mínimas no código fonte este formato pode ser modificado para o formato binário ou até mesmo XML.

Outra biblioteca utilizada na implementação do protótipo foi a coleção de classes PACC [36]. Esta biblioteca também foi desenvolvida em ANSI C++ de modo a ser portável entre diferentes plataformas. Ela disponibiliza classes para comunicação em rede (TCP / UDP) bem como classes para a criação e gerenciamento de *threads*, oferecendo assim uma abstração única tanto para ambiente Windows quanto para ambiente Unix. A HoloVM antes da concepção de Holo*Go* já utilizava esta biblioteca. Desta maneira toda a

comunicação relativa à mobilidade forte de código foi implementada utilizando as classes previamente disponibilizadas por esta biblioteca.

5.2 Funcionalidades Introduzidas no Ambiente

Visando facilitar o desenvolvimento de aplicações que utilizem mobilidade de código, três novas funcionalidades foram acrescentadas ao ambiente. A primeira delas compreendeu a concepção de um novo tipo de ente. A segunda implicou na adição de funcionalidades ao HNS e a terceira compreendeu a adição de dois novos comandos à Hololinguagem.

O novo tipo de ente é chamado de ente *holder*. Este ente tem apenas o papel de ficar parado esperando que entes entrem nele de modo a executar uma tarefa. A Figura 5.2 mostra o código fonte de um ente *holder*.

```
Holo()
{
    holo()
    {
      clone(holder,holder);
    }
}

holder()
{
    holder()
    {
      history#DONE;
    }
}
```

Figura 5.2 - Código fonte de um ente *holder*

O funcionamento de um ente deste tipo é bastante simples. Ao ser instanciado ele fica aguardando pela escrita de uma tupla contendo a *string* DONE em sua história. Enquanto isso não acontece, ele fica apenas aguardando e servindo como um lugar onde entes podem entrar de modo a executar tarefas.

No momento em que um ente deste tipo é instanciado em uma HoloVM, o HNS é notificado como ocorre na instanciação de qualquer ente, e adicionalmente atualiza uma lista com a localização de todos os entes *holder* presentes no ambiente distribuído. A localização de cada *holder* é mantida através do nome do *host* que o contém. Esta

informação é agregada ao nome deste ente de modo a diferenciar entes *holder* em diferentes locais (ambientes computacionais). Desta maneira uma aplicação Holo pode perguntar ao HNS quantos ambientes computacionais presentes no ambiente distribuído estão aptos a receber entes em um determinado momento.

Ambos os comandos adicionados à linguagem interagem diretamente com o HNS de modo a obter informações a respeito do ambiente de execução distribuído. A Tabela 5.1 descreve os comandos introduzidos.

Tabela 5.1 - Comandos adicionados à Hololinguagem

Comando	Descrição
getvms	**Retorna o número de entes *holder* em execução no ambiente distribuído.**
getfreevm	**Retorna o próximo ente *holder* que não contenha nenhum ente executando**

O comando **getvms** retorna o número de HoloVMs que contenham entes do tipo *holder* executando. O comando **getfreevm** retorna o nome de um ente *holder* que esteja livre. Na implementação atual, um ente *holder* pode estar livre (nenhum ente executando dentro dele) ou ocupado (pelo menos um ente executando dentro dele). Neste sentido o comando **getfreevm** tentará encontrar um ente *holder* que esteja livre e retornará seu nome para o programa Holo que o invocou.

Esta abordagem visa apenas criar uma infra-estrutura de software que torne mais fácil o desenvolvimento e também a validação de aplicações Holo que utilizem a mobilidade forte de código. Em uma implementação mais aprimorada o HNS poderia utilizar mais informações a respeito de cada ente *holder* de modo a explorar melhor a distribuição de entes entre ambientes computacionais. Entretanto uma abordagem deste tipo está fora do escopo deste trabalho, que está focado na validação do modelo de mobilidade forte de código proposto.

5.3 Resultados Experimentais

Quando da ocorrência de uma mobilidade de código, três elementos determinam o tempo necessário para que a mesma ocorra: tempo de serialização, tempo de envio à HoloVM de destino e tempo de deserialização. Sendo assim, experimentos foram realizados de modo a medir os tempos necessários para cada um destes elementos.

Inicialmente foram realizados os experimentos referentes aos tempos de serialização e deserialização. Neste sentido, um ente elementar foi criado para computar os tempos de serialização e deserialização. Este mesmo ente foi utilizado para os experimentos realizados com entes compostos. Nestes experimentos outros entes foram movidos para dentro dele antes da serialização. Os resultados obtidos neste experimentos são mostrados na Tabela 5.2.

Estes experimentos foram realizados em uma máquina com um processador Athlon XP 1.8 Ghz com 512 Mb de memória RAM utilizando o sistema operacional Windows XP Professional.

Tabela 5.2 - Tempos de serialização e deserialização

Número de entes filhos	Tempo de Serialização (s)	Tempo de Deserialização(s)	Desvio Padrão
0	**0,015**	**0,015**	**0,0008**
1	**0,016**	**0,016**	**0,0009**
2	**0,017**	**0,017**	**0,0009**
3	**0,018**	**0,018**	**0,0010**

Os resultados apresentados nas colunas dois e três são uma média de dez execuções. Comparando os tempos obtidos na serialização e deserialização de um ente elementar com os tempos obtidos com um ente composto contendo apenas um filho, observa-se um crescimento de apenas um milisegundo. No protótipo atual de Holo*Go*, em um ente composto, todos os entes componentes compartilham a mesma *Constant Pool*. Desta maneira, ao se mover um ente elementar ou um ente composto, somente uma instância da *Constant Pool* é movida. Além disso, verifica-se que o tempo de serialização aumenta linearmente na medida em que são acrescentados entes componentes.

Visando obter o tempo necessário para o envio de um ente a outro ambiente computacional, o mesmo ente utilizado no experimento anterior foi serializado, enviado a

outro ambiente onde foi deserializado e enviado de volta ao ambiente de origem. O esquema utilizado para esta medição é mostrado na Figura 5.3. O tempo necessário para completar a operação foi computado no ambiente computacional A, onde o ente foi instanciado e enviado ao ambiente computacional B. A fim de obter somente o tempo de envio, os tempos de serialização e deserialização em ambos os ambientes computacionais foram subtraídos do tempo total.

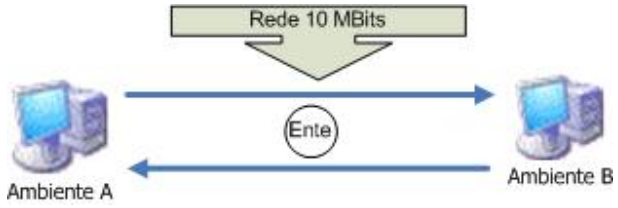

Figura 5.3 - Configuração utilizada no experimento

Para este experimento apenas duas máquinas foram utilizadas. Ambas com processador Athlon XP 2800 com 512 Mb de memória RAM utilizando o sistema operacional Windows XP Professional e conectadas através de uma rede ethernet de 10 Mbits.

A mesma abordagem do primeiro experimento foi utilizada. Inicialmente foi enviado um ente elementar. Em um segundo instante foi enviado um ente composto com apenas um filho. Em seguida o experimento foi repetido com entes compostos com dois e três filhos. Os resultados obtidos e mostrados na Tabela 5.3 são uma média de dez execuções e correspondem somente ao tempo necessário para o envio de um ente a outro ambiente computacional.

Tabela 5.3 - Tempo de envio de um ente

Número de entes filhos	Tempo (s)	Desvio Padrão
0	**0,010**	**0,002**
1	**0,015**	**0,002**
2	**0,020**	**0,003**
3	**0,025**	**0,002**

Através dos dados obtidos, observa-se uma tendência linear no tempo de envio dos entes. À medida que entes filhos são adicionados à hierarquia, o tempo cresce proporcionalmente. O ente utilizado neste experimento possui somente uma ação que

realiza o cálculo da série de Fibonacci. O gráfico da na Figura 5.4 mostra a tendência linear obtida neste experimento.

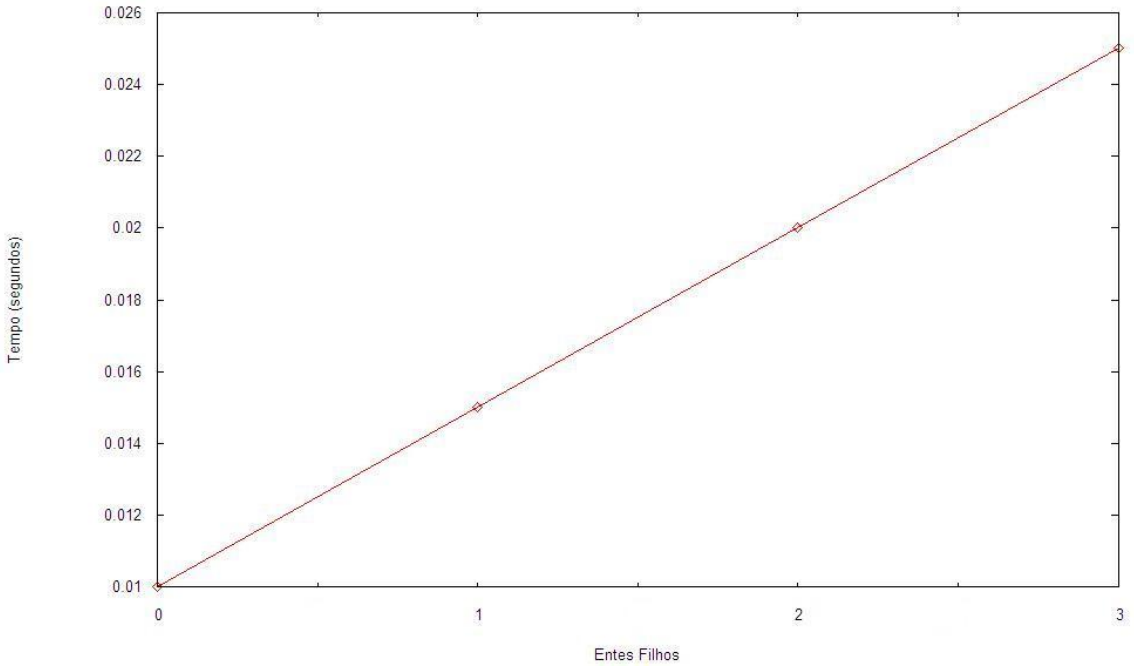

Figura 5.4 - Tempo de envio

O tempo pode variar de acordo com o número de ações contidas no ente, pois um ente é enviado a outro ambiente computacional na forma de uma seqüência de bytes. Neste sentido, quanto mais ações ou entes filhos um ente tiver, maior será sua seqüência de bytes e, consequentemente, maior será o tempo necessário para o seu envio.

5.4 Conclusão

Este capítulo apresentou detalhes específicos da implementação do Holo*Go* juntamente com resultados referentes aos passos de serialização, deserialização e envio de entes envolvidos em uma mobilidade de código. Holo*Go* foi desenvolvido inteiramente em ANSI C++ de modo a ser compatível com uma diversa gama de sistemas operacionais e dispositivos e também facilitando sua integração com a HoloVM. Funções de serialização e deserialização foram acrescentadas às estruturas envolvidas na execução de um ente. O uso das bibliotecas Boost e PACC facilitou o processo de desenvolvimento.

Tendo apresentado os detalhes específicos referentes à implementação do modelo juntamente com experimentos iniciais, o próximo capítulo apresenta aplicações desenvolvidas que utilizam mobilidade de código através das funcionalidades oferecidas por Holo*Go*.

6. Aplicações

Visando situar o modelo desenvolvido no contexto real de aplicações, neste capítulo serão apresentados cenários que se beneficiam da utilização de mobilidade de código. Para cada um dos cenários apresentados, um programa utilizando a Hololinguagem juntamente com Holo*Go*, foi desenvolvido e testado.

6.1 Ganho de Desempenho

O primeiro cenário modelado compreendeu uma aplicação que visa o ganho de desempenho na execução de uma tarefa computacionalmente intensa. Esta aplicação possui como entrada uma tarefa (denotada por uma matriz) a ser executada e, baseada nos recursos disponíveis no ambiente, esta tarefa é dividida em pedaços iguais. Cada pedaço da tarefa é atribuído a um ente que se move para um ente do tipo *holder* a fim de aplicar o processamento ao seu pedaço da tarefa. Neste sentido, o ente que se move carrega os dados (pedaço da tarefa) e também o código a ser aplicado. Utilizando esta abordagem um novo algoritmo pode ser adicionado à aplicação apenas inserindo uma nova ação no ente que o aplica. A Figura 6.1 mostra o código referente ao ente que se move para um ente do tipo *holder*.

```
1   Task()
2   {
3         Task(task,Target_Being)
4         {
5               move(self,Target_Being);
6               DoTask(task);
7               move (self,launch);
8         }
9
10        DoTask(task)
11        {
12              ...
13        }
14  }
```

Figura 6.1 - Trecho de **c**ódigo fonte da aplicação focada em ganho de desempenho

Inicialmente a aplicação requisita ao HNS a quantidade de entes do tipo *holder* disponíveis no ambiente. Baseado nesta informação a aplicação divide a tarefa a ser executada e para cada parte da tarefa um ente é instanciado e move-se para um dos entes *holder* do ambiente.

O código localizado entre as duas instruções *move* na linha 6 é executado em uma máquina remota. Ao ser clonado, o ente descrito na Figura 6.1 recebe como argumento o nome de um ente do tipo *holder* para onde ele deve se mover juntamente com o pedaço da tarefa a ser processada. O primeiro comando executado por este ente (linha 5) é um comando *move* que efetiva a mobilidade de código. Depois de executar a tarefa na máquina remota este ente volta para a máquina de origem através de outra chamada ao comando *move* (linha 7).

Os testes foram realizados com quatro tamanhos diferentes de entradas: 500x500, 1000x100, 2000x2000 e 4000x4000 respectivamente. Os resultados obtidos demonstram que a tendência manteve-se a mesma para todas as entradas. Executando a aplicação com apenas uma máquina disponível no ambiente e utilizando uma entrada de tamanho 1000x1000, o tempo médio de execução foi de 19.72 segundos. Na medida em que mais máquinas entraram no ambiente e anunciaram seus entes *holder* para o HNS, a aplicação pode perceber esta mudança e dividir a tarefa entre estas máquinas. Sendo assim, executando a aplicação com duas, três e quatro máquinas no ambiente os tempos médios obtidos foram 8.94, 6.50 e 5.70 segundos respectivamente.

A Figura 6.2 sumariza os resultados apresentados na Tabela 6.1 que foram obtidos executando os experimentos em um ambiente de testes. Três computadores Athlon XP com 512 MB de memória RAM com sistema operacional Windows XP juntamente com um computador Pentium IV também com 512 MB de memória RAM foram utilizados para realizar os experimentos. Um computador Core 2 Duo com 1 GB de memória RAM com sistema operacional Windows XP foi utilizado para o gerenciamento da aplicação. Cada um dos experimentos foi executado dez vezes.

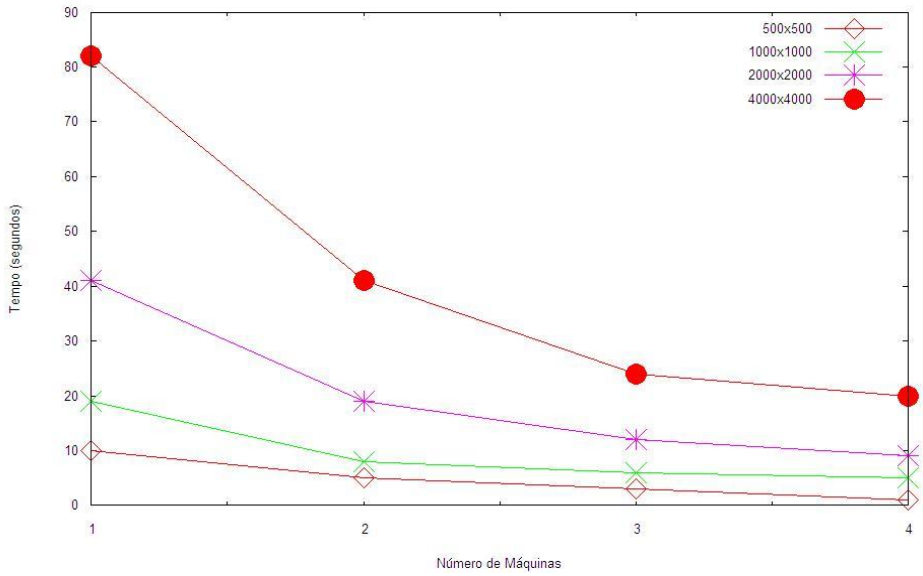

Figura 6.2 - Tempo de execução X Número de máquinas

Tabela 6.1 - Tempo de execução X Número de máquinas

	1		2		3		4	
	Média	D. P.	Média	D. P.	Média	D.P.	Média	D.P.
500x500	10.26	0.93	5.62	0.25	3.39	0.09	1.77	0.04
1000x1000	19.72	0.55	9.72	0.51	6.51	0.31	5.71	0.37
2000x2000	41.37	0.36	19.91	0.25	12.63	0.29	9.41	0.32
4000x4000	82.55	0.65	41.62	0.33	24.71	0.22	20.22	0.24

Existem outros fatores que influenciam no tempo total de execução como a latência da rede e o tamanho de um ente serializado. Neste experimento, entretanto, o foco principal é a funcionalidade do modelo de mobilidade de código desenvolvido. Neste sentido Holo*Go* mostrou-se uma alternativa viável para a redução do tempo total de execução de uma tarefa computacionalmente intensa e, ao mesmo tempo em que aproveitou os recursos oferecidos pelo ambiente.

6.2 Gerenciamento de Rede

A demanda por uma quantidade cada vez maior de informações de gerenciamento, aliada ao aumento dos requerimentos de confiabilidade exigidos por aplicações que executam em ambiente de rede torna a utilização de código móvel uma alternativa para este tipo de cenário. Atualmente o protocolo SNMP é um dos mais utilizados para a obtenção de informações gerenciais. Utilizando uma abordagem centralizada, questões referentes à

escalabilidade, flexibilidade e desempenho tornam-se prejudicadas [17]. Neste sentido, a utilização de agentes móveis para executar tarefas de gerenciamento de rede apresenta-se como uma possível solução para estes ambientes, pois estes são menos sensíveis à latência e também a largura de banda [17].

Com isso em mente o segundo cenário modelado apresenta um ambiente composto por diferentes máquinas, das quais se quer obter informações gerenciais. Neste sentido, a aplicação desenvolvida compreende um ente que viaja de modo a coletar informações a respeito de cada ambiente computacional por onde ele passa. Uma aplicação Holo instancia um ente que requisita ao HNS informações a respeito do ambiente. De posse destas informações o ente inicia sua viajem e, por cada ambiente computacional por onde ele passa, ele executa uma ação que pesquisa pelos recursos existentes. Esta informação a respeito dos recursos disponíveis em cada ambiente fica armazenada no ente que, ao retornar para a estação de origem leva estas informações para seu ente pai. A Figura 6.3 mostra o código fonte desta aplicação.

```
1  holo()
2  {
3        holo()
4        {
5              getvms(N);
6              for (X := 1 to N){
7                    getfreeVM(FreeVM);
8                    VMS[X] := FreeVM;
9              }
10             clone(agent(VMS),null);
11       }
12
13       agent()
14       {
15             agent(ListVMS)
16             {
17                   for X := 1 to listsize(ListVMS){
18                         move(self,ListVMS[X]);
19                         getresdata(Data);
20                         Vetor[X] := Data;
21                   }
22                   move(self,holo); //Retorna ao ambiente computacional
23                                    //de origem levando junto o vetor
24                                    //com os dados coletados
25             }
26       }
```

Figura 6.3 - Código fonte da aplicação de gerenciamento

Neste código, inicialmente um vetor é carregado com informações referente ao ambiente (linhas 5 a 9). Tendo este vetor carregado, a aplicação instancia o ente *agent* que irá viajar pelo ambiente. Este ente recebe como argumento o vetor contendo as máquinas para onde ele deve se mover. O código executado pelo ente agente (linhas 16 a 22) é um laço que itera pelo vetor que contém as máquinas do ambiente. Para cada máquina que o agente visita ele executa o comando `getresdata` que obtém os seguintes dados a respeito do ambiente computacional: endereço IP, número de processadores, velocidade do processador, quantidade de memória RAM e espaço livre em disco. À medida que o ente viaja ele coleta estas informações e armazena elas em um vetor (linha 20). Depois de viajar por todo o ambiente o ente retorna a máquina que o lançou com o vetor contendo em cada posição os dados referentes a cada máquina por onde ele passou. A Figura 6.4 mostra este cenário.

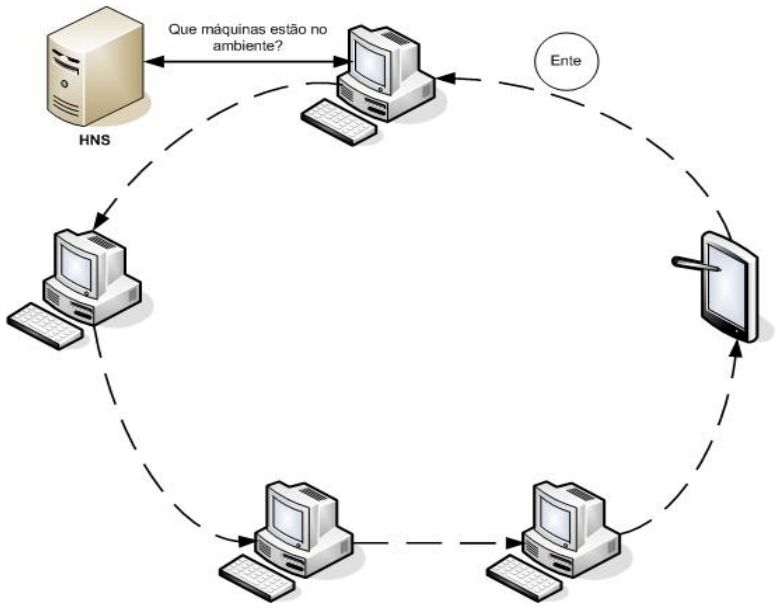

Figura 6.4 - Ambiente utilizado para testar a aplicação de gerenciamento de rede

Um ambiente de testes composto de cinco máquinas foi utilizado para a execução desta aplicação: três máquinas com processador AMD Athlon XP 2000 com 512 Mb de memória RAM, uma máquina com processador Intel core 2 duo com 1 Gb de memória RAM e um *Tablet* PC com processador Intel Centrino 1.5 Ghz com 512 Mb de memória RAM. O sistema operacional utilizado por todas as máquinas deste experimento foi o Windows XP Professional, com exceção do *Tablet* PC que utilizou o sistema operacional

Windows XP *Tablet* PC *Edition*. O conteúdo do vetor ao retornar a máquina que o lançou é mostrado na Tabela 6.2.

Tabela 6.2 - Conteúdo do vetor após o ente percorrer o ambiente

Índice	IP	Número de Processadores	Velocidade Processador	Memória Ram	Espaço livre em disco
1	**10.17.174.29**	**2**	**1830 MHz**	**1 Gb**	**64.8 Gb**
2	**10.17.174.208**	**1**	**1850 MHz**	**512 Mb**	**17.4 Gb**
3	**10.17.174.111**	**1**	**1850MHz**	**512 Mb**	**23.8 Gb**
4	**10.17.174.79**	**1**	**1850 MHz**	**512 Mb**	**12.4 Gb**
5	**10.17.174.51**	**1**	**1054 MHz**	**512 Mb**	**31.7 Gb**

Nesta aplicação o foco principal novamente foi a funcionalidade provida por Holo*Go*. Com poucas modificações esta aplicação pode ser utilizada para outros fins, como por exemplo, descoberta de recursos. Neste contexto um ente pode viajar pelo ambiente a procura de algum recurso específico necessário para a execução de alguma tarefa.

6.3 Computação Ubíqua

A mobilidade de código é uma parte importante no desenvolvimento de um sistema de computação ubíqua. Dentre suas principais utilizações encontram-se a adaptação dinâmica a ambientes inteligentes, a adição automática de funcionalidades a aplicações e a implementação da semântica *follow-me* [3], onde o programa viaja pelo ambiente juntamente com o usuário.

Com isto em mente uma integração com um projeto desenvolvido no laboratório de computação móvel[2] da Unisinos foi realizada. Este trabalho compreende a concepção de um servidor de localização, chamado SELIC (Servidor de Localização e Informação de Contexto). O SELIC é um servidor que permite que aplicações acessem informações sobre localização de dispositivos. A sua tarefa é integrar diversos sistemas de localização (GPS, RFID, IEEE 802.11, etc.) que possuem diferentes formatos de dados para fornecer a informação de posicionamento em uma interface de alto nível. Para isto ele disponibiliza uma API através de uma interface de *Web Services* de modo a facilitar a integração com outros sistemas. Uma adaptação neste servidor possibilitou que este exporte ao HNS,

[2] http://www.inf.unisinos.br/mobilab

através de sua API de *Web Services*, informações a respeito da localização física dos dispositivos que contenham entes *holder*.

A configuração deste experimento é mostrada na Figura 6.5. Esta figura mostra a planta baixa do segundo andar do prédio onde se encontra o laboratório.

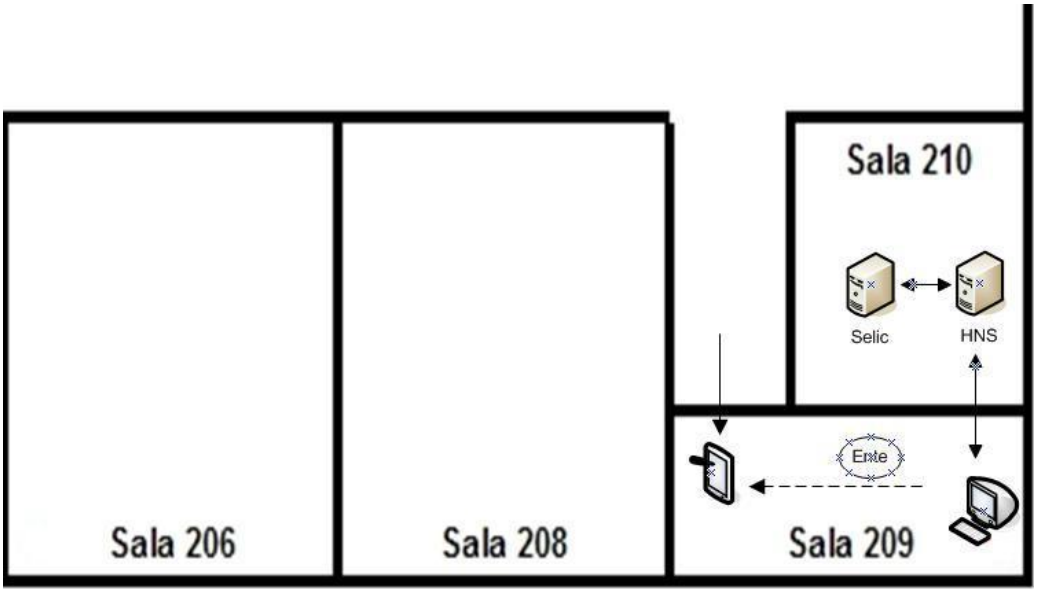

Figura 6.5 - Integração com servidor de localização SELIC

Uma aplicação Holo foi concebida de modo que, para qualquer dispositivo que contenha um ente *holder* e entre na sala 209, um ente é movido para dentro dele de modo a coletar informações a respeito dos recursos oferecidos pelo dispositivo que hospeda o ente. Para isto, esta aplicação fica perguntando ao HNS a respeito da localização dos entes. Caso algum ente *holder* encontre-se dentro da sala 209, esta aplicação executa uma mobilidade de código. O código fonte da aplicação que implementa este serviço é mostrado na Figura 6.6.

Também foi adicionado à Hololinguagem um comando (**`getlocationdata`**) que obtém informações a respeito da localização física dos entes *holder* em execução no ambiente. Quando este comando é executado, um vetor contendo a localização física de todos os entes *holder* presentes no ambiente é retornado à aplicação. De posse destas informações a aplicação pode, por exemplo, mover entes somente para dispositivos que estejam em determinada sala.

Na situação mostrada na Figura 6.5, um *Tablet* PC entrou na Sala 209 executando um ente *holder*. A informação de sua localização é passada ao servidor SELIC que a repassa para o HNS. Desta maneira, ao requisitar informação de localização, a aplicação que está executando descobre que este dispositivo encontra-se fisicamente localizado na Sala 209 e automaticamente executa uma mobilidade de código.

```
1  holo()
2  {
3        holo(Location)  //localização física passada como parâmetro
4        {
5              while(1){
6                    getlocationdata(Vet);
7                          for X := 1 to listsize(Vet) do {
8                                if (Vet[X] == Location)
9                                      clone(agent(Vet[X+1]),null);
10
11                               X := X + 1;
12                         }
13                   sleep(5000);
14             }
15       }
16 }
17
18 agent()
19 {
20       agent(Location)
21       {
22             move(self,Location);
23             getresdata(Data);
24             move(self,holo);
25       }
26 }
```

Figura 6.6 - Código fonte da aplicação ubíqua

Esta aplicação executa um laço infinito que a cada cinco segundos, através do comando **`getlocationdata`** (linha 6), carrega um vetor com o nome e a localização física (dispositivo) dos entes *holder* em execução no ambiente. Tendo este vetor carregado a aplicação itera por ele de modo a verificar se algum ente *holder* se encontra fisicamente na localização passada como parâmetro de entrada para a aplicação. Caso exista algum, o ente *agent* é instanciado através do comando clone (linha 9) , se move para o dispositivo (linha 22), executa o comando que obtém os dados referentes aos recursos do dispositivo e retorna ao ambiente computacional de origem com estes dados.

6.4 Conclusão

Este capítulo consolidou a utilização de Holo*Go* através do desenvolvimento de três aplicações que utilizam mobilidade de código. O principal objetivo neste sentido foi mostrar a utilização do protótipo em cenários reais onde a mobilidade de código é utilizada. Neste sentido Holo*Go* mostrou-se como uma alternativa para ganho de desempenho, gerenciamento de rede ou recursos e também como suporte a ambientes ubíquos.

Outro aspecto a ser considerado é a facilidade de desenvolvimento oferecida. A mobilidade forte de código ocorre de maneira transparente através do comando *move*, livrando assim o desenvolvedor da tarefa de salvar e restaurar o estado de execução de um ente. O próximo capítulo apresenta as conclusões e as contribuições deste trabalho juntamente com os trabalhos futuros.

7. Considerações Finais

Esta dissertação apresentou um modelo de mobilidade forte de código. Este modelo, chamado Holo*Go*, disponibiliza esta funcionalidade dentro do contexto do Holoparadigma. Até então somente a mobilidade lógica era suportada, estando a mobilidade de código apenas na especificação.

Inicialmente foram apresentados conceitos envolvendo a mobilidade de código com o intuito de adotar uma terminologia comum durante o restante do trabalho. Também foram apresentados sistemas que utilizam de alguma forma a mobilidade de código. A maioria dos sistemas apresentados disponibiliza apenas a mobilidade fraca de código. Outra característica comum entre os sistemas que apresentam mobilidade forte de código é a utilização de um *runtime* proprietário, o que acaba dificultando a implementação de mobilidade forte. Três paradigmas de desenvolvimento de software direcionados à mobilidade de código também foram apresentados.

A intenção ao apresentar estes conceitos foi introduzir de maneira sucinta o conceito de mobilidade de código enfatizando as questões relevantes dentro do contexto do desenvolvimento de software. A partir da definição destes conceitos foi apresentado o Holoparadigma juntamente com seu ambiente de execução distribuído, que serviram como base para o projeto e desenvolvimento de Holo*Go*.

Utilizando a HoloVM como plataforma de execução, Holo*Go* foi desenvolvido de modo a não forçar nenhuma modificação na Hololinguagem. Utilizando esta abordagem, programas Holo escritos anteriormente a Holo*Go* não necessitam de modificações. Outra característica do Holo*Go* é a facilidade de implementação de programas que utilizam mobilidade forte de código. Através do comando *move* a mobilidade de código ocorre de maneira automática. Holo*Go* se encarrega de salvar o estado de execução no ambiente computacional de origem e restaura-lo no ambiente computacional de destino.

A validação de Holo*Go* ocorreu através da implementação de um protótipo. Este protótipo possibilitou a realização de experimentos iniciais, onde se verificou a viabilidade da mobilidade de código, tanto em entes elementares quanto em entes compostos.

Após a implementação do protótipo, aplicações que utilizam mobilidade de código foram desenvolvidas utilizando Holo*Go*. O intuito no desenvolvimento destas aplicações foi situar o protótipo em cenários reais de aplicações. Neste sentido Holo*Go* mostrou-se uma alternativa para ganho de desempenho, gerenciamento de rede ou recursos e também em ambientes ubíquos .

Como principal contribuição, este trabalho especificou, implementou e validou a mobilidade forte de código no Holoparadigma. Antes do desenvolvimento de Holo*Go*, a mobilidade de código constava apenas na especificação. Com a realização de Holo*Go*, além da mobilidade forte, foi obtida também a mobilidade fraca. Um ente pode se mover por ambientes computacionais e, em cada ambiente por onde ele passa instanciar outro ente através do comando *clone*. Holo*Go* ainda possibilita o desenvolvimento de programas que utilizam mobilidade de código utilizando os três paradigmas de desenvolvimento apresentados: COD, REV e MA.

Como trabalhos futuros tem-se a questão referente ao gerenciamento de recursos. Na versão atual, o único recurso sendo tratado é a *Constant Pool* que sempre é movida junto com o ente ao ambiente computacional de destino. Dado que atualmente a Hololinguagem não oferece funções de acesso a recursos nativos do sistema operacional subjacente, tais como acesso a arquivos e funções específicas, o tratamento de recursos fica prejudicado. Uma vez que estas funcionalidades estejam implementadas na Hololinguagem, funções de tratamento destes novos recursos deverão ser adicionadas ao modelo. Além destas funções de tratamento, opções de como os recursos serão tratados pelos entes também devem ser oferecidas.

Adicionalmente ao gerenciamento de recursos, a questão referente a entes que estejam esperando por ações de outros entes antes de serem movidos também deve receber maior atenção. Na versão atual, o ente a ser movido, incondicionalmente aguarda pelo retorno das ações que estejam executando em outros entes antes de executar sua mobilidade

de código. Esta abordagem pode ocasionar atraso ou até mesmo uma *deadlock* em uma mobilidade de código. Neste sentido um tratamento mais sofisticado, que por questões de tempo, não foi implementado neste trabalho, deve ser oferecido por Holo*Go*. O desenvolvimento de mais aplicações que utilizam a mobilidade de código, situando Holo*Go* em mais cenários também é uma questão a ser mais trabalhada.

Referências Bibliográficas

[1] A.L. Murphy, G.P. Picco, and G.-C. Roman. **LIME: A *Middleware* for Physical and Logical Mobility**. In F. Golshani, P. Dasgupta, and W. Zhao, editors, Proc. of the 21st Int. Conf. on Distributed Computing Systems (ICDCS-21), pages 524.533, May 2001.

[2] Augustin, Iara; Yamin, Adenauer; Geyer, Claudio. **Requisitos para o projeto de aplicações móveis distribuídas**. VIII Cacic Congresso Argentino de Ciências de la Computición. Santa Cruz Argentina. 15-20 october 2001.

[3] Augustin, Iara. Yamin, Adenauer, Geyer, Cláudio. **Managing the Follow-me Semantics to Build Large-Scale Pervasive Applications**. 3rd International Workshop on *Middleware* for Pervasive and Ad-Hoc Computing. ACM / IFIP / USENIX. Grenoble, France. 2005.

[4] Barbosa, J. L. V. (2002). **Holoparadigma: Um Modelo Multiparadigma Orientado ao Desenvolvimento de Software Distribuído.** Tese (doutorado em ciência da computação), Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre. 213p.

[5] Barbosa, J. L. V. and Geyer, C. F. R. **Uma linguagem multiparadigma orientada ao desenvolvimento de software distribuído**. In Proceedings of the V Simpósio Brasileiro de Linguagens de Programação (SBLP), 2001.

[6] Bellfemine, Fabio, Caire Giovanni, Trucco, Tiziana, Rimassa Giovanni. **JADE Programmer's Guide**. 2003 Disponível em http://sharon.cselt.it/projects/jade/doc/programmersguide.pdf Acessado em 03/2006

[7] Bonnato, Daniel Torres; Barbosa, Jorge Luís Victória; Cavalheiro, Gerson Geraldo Homrich. **Um Suporte à Computação Pervasiva para o Holoparadigma.** VI Workshop em Sistemas de Alto Desempenho (WSCAD 2005), Rio de Janeiro, 2005

[8] Bonatto, Daniel Torres (2005). **HNS: Uma Solução para Suporte á Execução Distribuída Considerando Aspectos da Pervasividade**. Dissertação (mestrado em comutação aplicada) Programa Interdisciplinar em Computação Aplicada. Ciências Exatas e Tecnológicas. Universidade do Vale do Rio dos Sinos, 103p.

[9] Bray, J. and Struman, C.F., **Bluetooth Connect Without Cables**. Prentice-Hall, Inc. 2001

[10] Cabri, Giacomo, et. al. **Strong Agent Mobility for Aglets based on the IBM JikesRVM.** ACM Symposium on Applied Computing. Dijon, France. April 23-27 2006.

[11] Cardelli, L**. A Language with Distributed Scope.** Computing Systems; vol. 8, no. 1, pp 27-59, 1995.

[12] Chakravarti, Arjav J. ; Wang, Xiaojin; Hallstrom, Jason O. ; Baumgartner, Gerald. **Implementation of Strong Mobility for Multi-threaded Agents in Java.** Proceedings of the 2003 International Conference on Parallel Processing (ICPP'03) .

[13] Chetan, Shiva; Al-Muhtadi, Jalal; Campbell, Roy and Mikunas, Dennis M. **Mobile Gaia: A *Middleware* for Ad-Hoc Pervasive Computing**. IEEE Consumer Communications and Networking Conference, 2004.

[14] D. Gelernter. **Generative Communication in Linda**. ACM Computing Surveys, 7(1):80.112, Jan. 1985.

[15] Delamarco, Marcio ; Picco, Gian Pietro**. Mobile Code in .NET: A Porting Experience.** In Proceedings of the 6th International Conference on Mobile Agents (MA 2002), Barcelona (Spain). N. Suri ed. Lecture Notes on Computer Science vol. 2355. pp 16-31, October 2002.

[16] Du Bois, A.R; Trinder, P.W.; Loidl, H-W. **Towards Mobile Skeletons**. In Parallel Processing Letters, v. 15, n. 3, p. 273-288, 2005.

[17] Eid, Mohamed, et. al. **Trends in Mobile Agents**. Journal of Information and Practice on Information Technology. Vol. 37. No. 4. November, 2005.

[18] Eisenberg, Andrew. Et al. **SQL:2003 has been published**. ACM Sigmod Record. Volume 33. Issue 1. march 2004.

[19] F. Bellifemine, G. Caire, A. Poggi, G. Rimassa, **JADE – A White Paper** Sept. 2003. Disponível em http://jade.tilab.com/papers/2003/ WhitePaperJADEEXP.pdf. Acessado em 11/2006.

[20] Felmetsger, Viktoria; Vigna, Giovanni. **Exploiting OS-level Mechanisms to Implement Mobile Code Security**. Proceedings of the 10th International Conference on Engineering of Complex Computer Systems (ICECCS`05), 2005.

[21] Fuggetta, Alfonso; Picco, Gian Pietro; Vigna, Giovanni **Unsderstanding Code Mobility**. IEEE Transactions on Software Engineering. Vol. 24 Num. 5. pp 342-361, 1998.

[22] G.-C. Roman, A.L. Murphy, and G.P. Picco. **Coordination and Mobility**. In A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf, editors,

Coordination of Internet Agents: Models, Technologies, and Applications, pages 254.273. Springer, 2000.

[23] Garzão, A. S. and Barbosa, J. L. V. (2003). **Uma máquina virtual com suporte á concorrência, mobilidade e blackboards**. In XXIX Conferência Latinoamericana de Informática (CLEI), volume 24, La Paz. Universidad Mayor de San Andrés.

[24] Ghezi, C.; Vigna, Giovanni. **Mobile Code Paradigms and Technologies: A Case Study**. Proceedings of the first International Workshop on Mobile Agents (MA'97), pp 39-49, LNCS 1219 Springer Verlag, Berlim, Germany, April 1997.

[25] Gosling, J.; Steele, Joy G; **The Java Language Specification.** Addison-Wesley, 1996.

[26] Grimm, R. **One.world : Experiences with a Pervasive Computing Architecture.** On Pervasive Computing, IEEE. pp 22-30. ISSN: 1536-1268. July-Sept 2004.

[27] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. **A note on distributed computing**. Technical Report TR-94-29, Sun Microsystems Laboratories, November 1994.

[28] **Jini Web page**. http://www.sun.com/jini . Acessada em 12/2006.

[29] Karlsson, Bjorn. **Beyond the C++ Stardard Library : An Introduction to Boost**. Addison-Wesley Professional, 432 p. New York, 2005.

[30] Lange, D. and Oshima, M. **Programming and Deploying Java Agents with Aglets.** Addison-Wesley. New York, 1998.

[31] Mockapetris, Paul V and Dunlap, Kevin J. **Development of the Domain Name System**. In SIGCOMM, pages 123-133, 1988.

[32] Naseen, Muhammad Kamram et al. **Implementing Strong Code Mobility**. In Information Technology Journal 3(2):188-191, 2004

[33] Neely, Mat. **Write Mobile Agents in .NET to Roam And Interact On Your Network.** MSDN Magazine online. http://msdn.microsoft.com/msdnmag/issues/06/02/MobileAgents/default.aspx . Acessado em 11/2006.

[34] Nork, Michael. **Voyager: An Overview**. Recursion Software Inc. http://www.recursionsw.com/Voyager/Voyager_High_Level_Overview.pdf Acessado em 03/2006.

[35] Object Management Group**. CORBA: Architecture and Specification** Aug. 1995

[36] Parizeau, Marc. **The PACC Collection** http://manitou.gel.ulaval.ca/~parizeau/PACC Acessado em Dezembro de 2006.

[37] Picco, Gian Pietro. **µCode: A Lightweight and Flexible Mobile Code Toolkit.** International Workshop on Mobile Agents. MA, 2, 1999. Stuttgart, Germany. Proceedings… Berlim: Springer, 1999. p 160-171 (Lecture Notes in Computer Science, v.1477).

[38] Sakamoto, T.; Sekiguchi, T. ; Yonezawa, **A. Bytecode Transformation for Portable Thread Migration in Java**. In Proceedings of Agent Systems, Mobile Agents , and Applications, 2000.

[39] Sekiguchi, T.; Masuhara, H; Yonezawa, A,. **A Simple Extension for Java Language for Controllable Transparent Migration and its Portable Implementation.** In Coordination Models and Languages, 1999.

[40] Silva, Luciano Cavalheiro da (2003). **Primitivas para Suporte à Distribuição De Objetos Direcionadas a Pervasive Computing** . Dissertação. (mestrado em ciência da computação) Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, 123 p.

[41] Sousa, João Pedro; Garlan, David **Aura: An Architectural *Framework* for User Mobility in Ubiquitous Computer Environments.** Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture. Kluwer Academic Publishers, pp 29-43, August 2002.

[42] Thorn, Thommy. **Programming Languages for Mobile Code**. ACM Computing Surveys. Vol. 29. Num. 3. pp. 213-239, 1997.

[43] Torres, Daniel; Kellerman, Felipe; Barbosa, Jorge Luis V.; Ramos, José Dirceu and Cavalheiro, Gerson G. H. **Estratégias para localização em um ambiente de computação móvel**. In XXIX Conferência Latinoamericana de Informática (SEMISH), São Leopoldo, 2005. Unisinos.

[44] Bender, James C. **Voyager and Software Agents Applied to Simulation-Based Training and Exercise Management.** Recursion Software Inc. December, 2004

[45] Vraned, S and Stanojevic, M. **Integrating multiple paradigms within the blackboard**. IEEE Transactions on Software Engineering, March 1995.

[46] Xiao, Yang. Rosdahi, Jon. **Wireless Home Networks: Performance Analysis and Enhancement for the Current and Future IEEE 802.11 Mac Protocols**. Proceedings of the 9th Annual International Conference on Mobile Computing and Networking. ACM Press, 2003.

[47] Yamin, Adenauer C.; Augustin, Yara; Barbosa, Jorge Luis Victória; Silva, Luciano Cavalheiro da; Cavalheiro, Gerson Geraldo Homrich; Geyer, Claudio Fernando Resin. **A *Framework* for Exploiting Adaptation in High**

Heterogeneous Distributed Processing. In Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2002) pages 125-132, Los Alamitos ,2002. IEEE Computer Society.

[48] Waldo, Jim. **The Jini Network Centric Architecture.** Communications of the ACM. Vol 42. No 7. July 1999.