FUNDAÇÃO DE "ENSINO EURÍPIDES SOARES DA ROCHA"
CENTRO UNIVERSITÁRIO "EURÍPIDES DE MARÍLIA" - UNIVEM
PROGRAMA DE MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

LEONARDO SOUZA SILVA

Segurança em Código Móvel no Ambiente μCode

MARÍLIA/SP
2004

LEONARDO SOUZA SILVA

Segurança em Código Móvel no Ambiente μCode

Dissertação apresentada ao Programa de Mestrado do Centro Universitário Eurípides de Marília, mantido pela Fundação de Ensino Eurípides Soares da Rocha, para obtenção do título de Mestre em Ciência da Computação.

Orientador:
Prof Dr. Márcio Eduardo Delamaro.

MARÍLIA/SP

2004

SILVA, Leonardo Souza.
Segurança em Código Móvel no Ambiente μCode / Leonardo Souza Silva; orientador: Márcio Eduardo Delamaro. Marília/SP:[s.n.], 2004

109 f.

Dissertação (Mestrado em Ciência da Computação) - Centro Universitário Eurípides de Marília - Fundação de Ensino Eurípides Soares da Rocha.

1.Códigos Móveis. 2.Segurança.

CDD: 12004.12

LEONARDO SOUZA SILVA

Segurança em Código Móvel no Ambiente μCode

Banca Examinadora da dissertação apresentada ao Programa de Mestrado em Ciência da Computação do Centro Universitário Eurípides de Marília - UNIVEM, mantido pela Fundação de Ensino Eurípides Soares da Rocha, para obtenção do título de Mestre em Ciência da Computação.

Resultado: **APROVADO**.

Marília/SP, 28 de Outubro de 2004.

Se não fosse pelo amor, amizade, cumplicidade e a dedicação de minha família, a quem muito admiro e tenho orgulho, com certeza este trabalho não seria concluído.

Dedico-o a vocês

(meu pai Laerte, minha mãe Maria do Rosário e minhas irmãs Consuelo e Anahí).

AGRADECIMENTOS

A conclusão deste trabalho só foi possível devido ao apoio de várias pessoas, as quais deixo registrado aqui, os meus mais sinceros agradecimentos. Após percorrer (literalmente) alguns milhares de quilômetros até este momento, algumas pessoas tornaram-se especiais nesta jornada, seja pelo apoio, pela motivação ou mesmo pelo exemplo dado no dia-a-dia, em especial deixo aqui registrado meu agradecimento:

- Ao prof. Dr. Márcio Eduardo Delamaro, meu orientador, a quem agradeço pela compreensão, pelos ensinamentos e principalmente pelo exemplo de profissionalismo e competência. Assim como, também agradeço aos professores da UNIVEM: Marcos Mucheroni, Jorge, Fátima e Dino, pelas constantes palavras de apoio, pelos conselhos e orientações.

- As professoras Conceição Butera e Ivanilde Herrero Fernandes Saad, da Universidade Católica Dom Bosco - UCDB, e aos professores Karen Kiomi Nakazato e José Craveiro da Costa Neto, da Universidade Federal de Mato Grosso do Sul - UFMS, pela amizade, incentivo e apoio, e por terem despertado em mim o interesse pela docência e pela pesquisa.

- Ao grande amigo Mauro Conti Pereira, com quem tenho o prazer de trabalhar na Universidade Católica Dom Bosco - UCDB, pela amizade, orientações e exemplo de dedicação e competência.

- Agradeço aos meus colegas no Laboratório de Teste de Software - LoST/UNIVEM, Gustavo e Danda, parceiros de estudo no intrínseco mundo do ambiente μCODE. Em especial, agradeço ao Rodrigo Fraxino de Araújo, aluno de iniciação científica, que muito me ajudou na implementação deste trabalho.

- Agradeço aos *Fora da Curva* (Cláudia, Piva, Rosiane, Gislene, Luís, Adriane, Danda e Lucilena), por me lembrarem do valor e da importância da verdadeira amizade e companherismo. O curso jamais teria sido o mesmo sem a presença de vocês.

- Por fim, agradeço ao Marcos Augusto de Rossi Piva, "o Piva", a quem tenho a honra de ter como amigo e parceiro de longas horas de café e conversa. Meu agradecimento mais que especial a sua família, que tantas vezes me acolheu na cidade de Marília.

"... o verdadeiro Mestre é aquele que cria discípulos ..."

Paulo Freire

SILVA, Leonardo Souza. **Segurança em Código Móvel no Ambiente μCode**. 2004. 109 f. Dissertação (Mestrado em Ciência da Computação) - Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília/SP, 2004.

RESUMO

Mobilidade de Código não constitui exatamente em um novo conceito. Contudo, sua aplicação na concepção e no desenvolvimento de sistemas distribuídos voltados às redes globais ou de grande alcance, como a WWW, além de ser recente, tem revolucionado a maneira como essas aplicações são concebidas e implementadas. Apesar dos benefícios advindos, dentre os quais destacamos a criação de sistemas mais configuráveis, considera-se que o potencial da aplicação da mobilidade de código ainda não foi plenamente explorado, estando restrito a alguns poucos *applets* Java executados através da Internet. Dentre os fatores que contribuem para esse cenário, estão as questões e implicações relacionadas à segurança de tais aplicações, que por vezes tornam-se cruciais na adoção das soluções baseadas em agentes móveis, freqüentemente inviabilizando sua aplicação. Neste cenário emerge o μCODE, um ambiente para desenvolvimento de sistemas de código móvel, criado em linguagem Java, e que se diferencia de seus pares pelo seu foco centrado nos mecanismos que suportam a mobilidade de código. Deste modo, além de um ambiente voltado ao trabalho com código móvel, ele também contempla o uso de diferentes paradigmas de mobilidade. Incorporando ao ambiente original os recursos de Assinatura de Código e Permissões, disponibilizados pela Linguagem Java, este trabalho implementa um mecanismo explicíto de segurança, dando origem a uma versão segura do ambiente μCODE. Em seu atual estágio, esta versão segura, denominada μCODE Seguro, permite aos usuários efetivamente controlar o comportamento das aplicações utilizadas, através da atribuição de permissões aos usuários, além de possibilitar a proteção dos servidores contra os chamados agentes maliciosos.

Palavras-chave: Códigos Móveis. Segurança.

SILVA, Leonardo Souza. **Segurança em Código Móvel no Ambiente μCode**. 2004. 109 f. Dissertação (Mestrado em Ciência da Computação) - Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília/SP, 2004.

ABSTRACT

Code Mobility is not exactly a new concept. However, its application on the conception and development on distributed systems for global or wide area networks, such as the WWW, besides being quite new, has been causing a revolution on the way that those applications are conceived and implemented. Despite the upcoming advantages, such as the creation of more configurable systems, we have to consider that the potential application of the code mobility has not been yet fully exploited, being restricted to a few Java *applets* executed through the Internet. Among the factors that contributes to that scenario, there are issues and implications related to those applications security, which sometimes become crucials to the solutions adopted based on mobile agents, which frequently make those applications not viable. In this scenario emerges the μCODE, an environment for mobile code systems created in Java, which differs from its pairs by its focusing centered on the mechanisms that support code mobility, and therefore, besides the environment focused on mobile code, it also includes the usage of new mobility paradigmas. Through the incorporation of code signing and permissions, available in Java language, this work implements a explicit security mechanism, creating a safe version of μCODE environment. In the current stage, the version called μCODE Safe, allow users to control the behavior of used applications and protect servers against malicious agent.

Keywords: Mobile Code. Security.

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

Com a popularização da World Wide Web - WWW, dá-se início uma demanda por sistemas que apresentam um grau cada vez maior de complexidade. Devido a carência de métodos específicos, tais aplicações têm sido desenvolvidas de acordo com as tradicionais técnicas para construção de sistemas distribuídos. Como reflexo, observa-se que tais aplicações, fruto muitas vezes da migração dos chamados sistemas legados, não tem oferecido o grau de flexibilidade e configurabilidade desejado em um ambiente tão heterogêneo.

Este fato tem fomentado o surgimento de diversas abordagens ao desenvolvimento de sistemas distribuídos voltados à grande rede, sendo aqui destacada a aplicação dos conceitos relacionados à Mobilidade de Código na construção dessas aplicações. A utilização de códigos móveis no desenvolvimento de aplicações para a WWW, na forma de sistemas: bancários *on-line*, de reserva de passagens, entre outros, tem se mostrado muito proveitosa por produzir sistemas cada vez mais adaptáveis ao ambiente da Internet.

Contudo, a adoção de soluções baseadas na utilização da Mobilidade de Código por vezes são inviabilizadas em função das implicações relacionadas à Segurança, que devem abranger tanto as próprias aplicações quanto os ambientes onde são executadas. Em geral, esse tipo de aplicação está associada a recepção e execução de códigos oriundos de locais, muitas vezes, externos ao domínio dos usuários. Mesmo se considerada a execução da aplicação apenas no domínio do usuário, faz-se necessário o estabelecimento de políticas e mecanismos que permitam realizar o controle das ações que cada usuário possui, ou não, permissão para executar.

1.1 Justificativa

A utilização da migração de código no desenvolvimento de sistemas distribuídos para redes globais[1], ou de grande alcance como a *World Wide Web* - WWW, tem se mostrado uma das mais promissoras alternativas na resolução de vários problemas que são enfrentados pelos desenvolvedores de sistemas distribuídos voltados a esse tipo de rede de computadores (FUGGETTA; PICCO; VIGNA, 1998).

Segundo Fuggetta *et al.* (FUGGETTA; PICCO; VIGNA, 1998), e também de acordo com Picco (PICCO, 2001a), as tecnologias, arquiteturas e metodologias, tradicionalmente empregadas no desenvolvimento de sistemas distribuídos, em sua grande maioria falham quando aplicadas a um ambiente que não o de uma rede local[2], justamente por não oferecerem o grau necessário de flexibilidade, customização, escalabilidade e configurabilidade. O que em parte pode ser explicado pelo fato dos sistemas distribuídos tradicionais assumirem a configuração estática do ambiente onde suas aplicações são executadas.

Em termos gerais, na abordagem tradicional a aplicação é construída e particionada, posteriormente é decidido onde cada parte deve ser executada. Em contra-partida, com a utilização da mobilidade de código, o código tem condições de movimentar-se entre os nós de uma rede, tornando possível a configuração dinâmica dos sistemas distribuídos (TANENBAUM; STEEN, 2002).

Essa mudança, foi desencadeada pelos avanços tecnológicos alcançados nos últimos anos que possibilitaram a introdução do chamado grau de mobilidade nos sistemas distribuídos. Apesar de não ser um conceito novo aos pesquisadores, uma vez que está presente em áreas como Sistemas Operacionais através da migração de processos, pouco se sabia sobre a possibilidade de transferir além de dados, programas, sendo que em algumas situações estes programas ainda encontram-se em execução.

No final da década de 90, o campo de pesquisa sobre Mobilidade de Código emergiu e inúmeros trabalhos foram iniciados na tentativa de consolidar esse paradigma. Para Fuggetta *et al.* (FUGGETTA; PICCO; VIGNA, 1998), a migração de código pode ser classificada sob três dimensões: Tecnologias, Paradigmas de Projeto e Domínio das Aplicações. Destaca-se aqui, que um dos maiores focos de atenção dos pesquisadores tem sido as tecnologias, sendo que várias propostas de linguagens para código móvel surgiram

[1] Do inglês: WAN - *Wide Area Networks*
[2] Do inglês: LAN - *Local Area Networks*

ao longo dos anos, oferecendo diferentes abordagens e mecanismos de suporte à mobilidade de código. Dentre as quais podemos destacar: a linguagem Java[3], Java *Aglets*[4], Telescript[5] e *Agent Tcl*[6] ou *D´Agents*, entre outros. Em função das suas características nativas, a linguagem Java é a base da implementação de muitas linguagens para código móvel, que por meio da definição de API´s (*Application Program Interface*), implementam seus mecanismos de suporte a mobilidade de código.

Apesar dos benefícios alcançados com a utilização de código móvel no projeto de aplicações distribuídas, sua utilização ainda está restrita a alguns poucos *applets* Java sendo executados através da WWW. Parte dessa sub-exploração se deve às preocupações com as questões relacionadas à segurança de tais aplicações, reflexo do enfoque centrado nas tecnologias, o que contribuiu para que grande parte dos ambientes voltados ao uso de mobilidade de código não tivessem mecanismos de segurança.

Neste cenário destacamos o ambiente μCODE (PICCO, 1998, 2001b), implementado na forma de uma API Java, também se apresenta como opção ao desenvolvimento de sistemas de código móvel. Seu diferencial entre as demais tecnologias é em função do enfoque dado aos mecanismos de mobilidade de código, o que permite ao programador optar pelo uso de diversas abordagens na implementação das aplicações.

Atualmente, o μCODE também não apresenta nenhum mecanismo explícito de segurança, o que motiva neste trabalho a busca pela definição e implementação de mecanismos de segurança para o ambiente μCODE, com base nos recursos suportados pela linguagem Java, na qual se baseia o ambiente. Espera-se com a implementação desse mecanismo oferecer condições para que as aplicações tenham condições de serem verificadas e controladas, reduzindo assim as possibilidades de ataques, roubo de informações, invasões, entre outros.

1.2 Objetivo

Em função do impacto negativo que a carência de mecanismos de segurança provoca na adoção de soluções baseadas na utilização da mobilidade de código, em especial

[3] *Sun Microsystems*: http://java.sun.com
[4] *IBM, Tóquio*: http://www.trl.ibm.com/aglets
[5] *General Magic*: http://reinsburgstrasse.dynds.org/mal/preview/telescript.29263.txt.html
[6] *University of Dartmouth*: http://agent.cs.dartmouth.edu

no contexto dos sistemas distribuídos voltados a ambientes heterogêneos como a WWW, objetiva este trabalho o estudo e a compreensão dos conceitos envolvidos, visando a definição e implementação de um mecanismo de segurança para o ambiente μCODE, de forma a permitir que seus usuários tenham condições de verificar e controlar o funcionamento das aplicações executadas por tal ambiente.

Para tanto, além da compreensão dos principais conceitos relacionados à Mobilidade de Código, é objetivo deste trabalho:

- A definição de uma versão segura para o ambiente μCODE, com base nos recursos de Assinatura de Código e Permissões, através da adequação e implementação desse ambiente, aos mecanismos de segurança definidos, dando origem ao chamado **μCode Seguro**.

1.3 Organização do Trabalho

O presente trabalho é composto de 6 capítulos, sendo os fatores que motivam a realização deste trabalho foram apresentados no primeiro capítulo.

O Capítulo 2, dá início a revisão bibliográfica por meio de um levantamento dos principais conceitos envolvidos na área de Mobilidade de Código, são apresentadas as formas de mobilidade e as implicações relativas à segurança. Destaca-se também o ambiente μCODE - em especial os critérios aplicados durante seu projeto, sua estrutura e funcionamento.

Após uma breve revisão dos principais conceitos da Linguagem Java, o Capítulo 3 descreve os mecanismos de segurança suportados pela linguagem, destacando os seus componentes e sua evolução.

O Capítulo 4 apresenta a implementação de uma versão segura do Ambiente μCODE, através do uso de recursos como Assinatura de Código e Permissões, com destaque as alterações realizadas e a forma como este mecanismo funciona. Os experimentos realizados são descritos e comentados no Capítulo 5.

Por fim, as conclusões e impactos esperados com a criação do ambiente **μCode Seguro** são apresentadas no Capítulo 6.

2 *Mobilidade de Código*

Tipicamente os sistemas distribuídos assumem a configuração estática dos ambientes onde as aplicações são executadas, dessa forma, a comunicação entre um conjunto de máquinas é realizada por meio de vínculos físicos cuja configuração é fixa e estática, ou seja, as aplicações são preparadas para serem executadas em determinados pontos de uma rede de computadores. Contudo, os avanços tecnológicos estão alterando essa visão por meio da introdução nos sistemas distribuídos do chamado grau de mobilidade - *degree of mobility* (PICCO, 2001a).

Segundo Picco (PICCO, 2001a), os graus de mobilidade podem ser divididos em: **Mobilidade Física** (*physical mobility*) - aquela onde a comunicação entre as máquinas de uma rede se mantém inalterada mesmo quando os nós dessa rede estão em movimento dentro de um espaço físico e **Mobilidade Lógica** (*logical mobility*) - aquela onde pode ocorrer em tempo de execução a migração total, ou mesmo parcial, do código e provavelmente do estado de execução desse código para outro nó da rede.

Devido à popularização da Internet, a Mobilidade Lógica, freqüentemente referenciada na literatura pelo termo **Mobilidade de Código**[1] (PICCO, 2001a; FUGGETTA; PICCO; VIGNA, 1998; TANENBAUM; STEEN, 2002), tem recebido considerável atenção por parte dos pesquisadores, que vislumbram em sua exploração a solução dos problemas de desempenho normalmente enfrentados pelos sistemas distribuídos de grande escala (TANENBAUM; STEEN, 2002).

As abordagens utilizadas na construção dos sistemas distribuídos tradicionais freqüentemente mostram-se ineficazes quando aplicadas em problemas que exigem distribuição em larga escala, como é o caso da Internet. Neste cenário, emergem as chamadas Linguagens de Código Móvel - MCL's[2] e são propostos diversos paradigmas que visam

[1] Do inglês: *Code Mobility*
[2] Do inglês: *Mobile Code Languages*

a explorar mobilidade de código, independentemente da tecnologia empregada (CARZANIGA; PICCO; VIGNA, 1997).

Além de apresentarem-se os conceitos básicos relativos à Mobilidade de Código, são comentadas nas próximas seções as tecnologias envolvidas e as questões de segurança associadas a esse tipo de aplicação.

2.1 Conceitos Básicos

Durante muitos anos a comunicação dos sistemas distribuídos esteve limitada à transferência de dados. Contudo, a evolução dos sistemas possibilitou a criação de situações em que transferir programas, algumas vezes enquanto são executados, tem simplificado o projeto de sistemas distribuídos, em especial os voltados às redes de grande alcance como a WWW (TANENBAUM; STEEN, 2002).

O conceito de processo origina-se do campo de Sistemas Distribuídos, sendo geralmente definidos como um programa em execução (TANENBAUM; STEEN, 2002), e a sua movimentação entre diferentes máquinas de uma rede tem sido objeto de muitos estudos, uma vez que essa migração é de grande valia no tratamento de questões como escalabilidade, balanceamento de carga entre nós de uma rede, na configuração dinâmica de clientes e servidores, além de melhorar a performance das aplicações por meio da exploração do paralelismo. As idéias envolvidas na transferência de fragmentos de código entre os nós de uma rede de computadores não é algo recente, estando a muito tempo presente na área de Sistemas Distribuídos na forma de migração de processo. Em um senso mais amplo, migração de código trata da movimentação de programas entre máquinas, com a intenção de ter determinados programas sendo executados em um, ou mesmo, em vários destinos. Assim como ocorre na migração do processo, em algumas situações é desejável que o *status* relativo à execução de um programa, os seus sinais pendentes, e outras partes relacionadas ao ambiente também sejam movidas.

De acordo com Fugetta *et al.* (FUGGETTA; PICCO; VIGNA, 1998), um processo é formado por três segmentos:

- Segmento de Código *(Code Segment)* - é a parte que contém o conjunto de instruções que compõem o programa que está sendo executado.

- Segmento de Recursos *(Resource Segment)* - contém as referências aos recursos externos necessários pelos processos, tais como: arquivos, impressoras, dispositivos, outros processos, entre outros.

- Segmento de Execução *(Execution Segment)* - é usado para armazenar o estado atual da execução de um processo, consistindo de: dados privados, a pilha e o contador de programas.

Segundo Tanenbaum (TANENBAUM; STEEN, 2002), há dois modelos para realização da migração de código, a forma mais simples é provida pela chamada **Mobilidade Fraca** (*Weak Mobility*) que somente a transferência do segmento de código. Sua principal característica está no fato de o programa transferido ser sempre iniciado, na máquina de destino, a partir de seu estado inicial - isto é o que acontece por exemplo, com um *applet* Java. Sua grande vantagem reside na simplicidade, uma vez que é requerida da máquina de destino apenas capacidade de executar o código.

Em contraste à Mobilidade Fraca, encontramos sistemas que oferecem suporte à **Mobilidade Forte** (*Strong Mobility*) sendo que neste caso,, além do código, o segmento de execução pode ser também transferido para a máquina destino. A característica típica desse modelo é que o processo em execução pode ser parado e posteriormente deslocado para outra máquina onde então retorna sua execução a partir do ponto onde parou na máquina de origem.

Apesar de a Mobilidade Forte ser um modelo muito mais poderoso que a Mobilidade Fraca, sua implementação é muito mais difícil, sendo suportada apenas por algumas tecnologias de mobilidade de código, dentre as quais destacamos: *Agent Tcl* ou *D' Agents* (*University of Darthmouth*) e *Sumatra* (*University of Maryland*), entre outras (TANENBAUM; STEEN, 2002).

O interesse pelas diferentes abordagens de Mobilidade de Código resultaram em um novo e promissor campo de pesquisa, cujo objetivo é explorar a noção de Código Móvel. Código Móvel é aquele que pode ser definido como um software habilitado a viajar através de uma rede de computadores heterogênea, atravessando domínios de segurança e sendo automaticamente executado no seu destino. Fuggetta *et al.* (FUGGETTA; PICCO; VIGNA, 1998), informalmente definem Mobilidade de Código como a capacidade de dinamicamente alterar a relação entre o fragmento de código e o local onde ele é executado.

A possibilidade de uso de componentes que dinamicamente possam alterar sua localização tende a causar a redefinição da estrutura lógica de sistemas distribuidos. Contudo, vale ressaltar que por ser um campo de pesquisa relativamente novo, ainda há uma carência por termos, definições e mesmo suporte metodológico relacionado à Mobilidade de Código, o que em várias situações dificulta sua aceitação.

As técnicas de migração de processo podem ser consideradas o marco inicial para uma nova categoria de sistemas, que oferecem formas avançadas de mobilidade de código. Esses sistemas são frequentemente referenciados como *Sistemas de Código Móvel* - MCS[3], sendo que exibem várias inovações em relação as abordagens existentes (FUGGETTA; PICCO; VIGNA, 1998):

- Mobilidade de Código é explorada em grande escala, a WWW;
- Mobilidade de Código não é influenciada pela localização dos componentes;
- A Mobilidade de Código está sob o controle do programador;
- A Mobilidade de Código não é executada apenas para balanceamento de carga.

Fuggetta *et al.* (FUGGETTA; PICCO; VIGNA, 1998), classifica a mobilidade de código em três dimensões que são de grande importância para o processo de desenvolvimento, uma vez que permitem que pesquisadores e usuários tenham como medir e avaliar diferentes soluções com respeito a um conjunto comum de conceitos e abstrações.

As três dimensões propostas por Fuggetta *et al.* (FUGGETTA; PICCO; VIGNA, 1998), são:

- **Tecnologias** - As tecnologias de Mobilidade de Código são as linguagens e sistemas que provêm mecanismos que permitem o suporte à mobilidade de código. São usadas no estágio de implementação.
- **Paradigmas de Projeto** - Estilos de arquitetura que podem ser usados pelo projetista na definição da arquitetura da aplicação. Esse estilo identifica uma configuração específica para componentes do sistema e suas interações mútuas, exemplos: cliente/servidor e ponto a ponto.

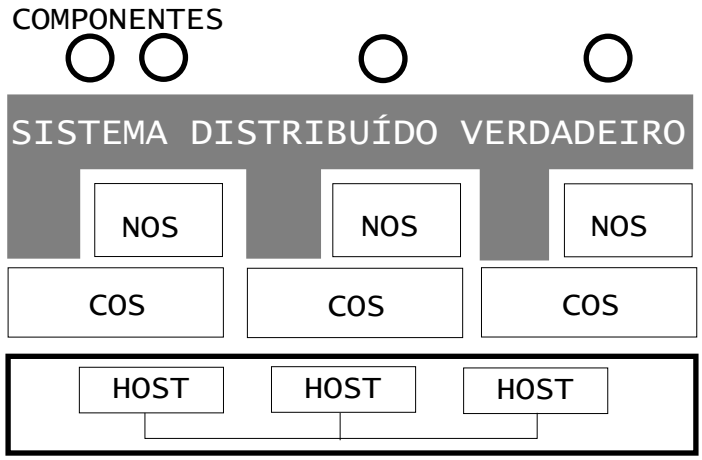
[3]Do inglês: *Mobile Code Systems*

- **Domínios de Aplicação** - Classes de aplicações que dividem o mesmo objetivo geral, por exemplo: Comércio Eletrônico e a Consulta de Aplicações Distribuídas - eles definem um papel na definição dos requisitos da aplicação.

A seguir, cada uma dessas três dimensões são discutidas mais detalhadamente.

2.2 Tecnologias de Mobilidade de Código

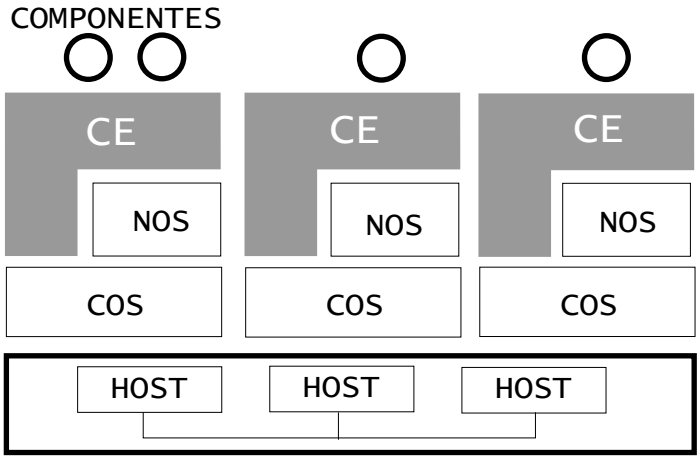
As tecnologias de mobilidade de código compreendem as linguagens de programação e seus correspondentes suportes em tempo de execução. Diferentemente da estrutura dos Sistemas Distribuídos Tradicionais, os Sistemas de Código Móvel apresentam uma estrutura básica de rede que não é escondida do programador (FUGGETTA; PICCO; VIGNA, 1998), conforme mostram as Figuras: 1 e 2.



NOS - Sistema Operacional de Rede; COS - Núcleo do Sistema Operacional; HOST - Máquina pertencente a rede.

Figura 1: Sistemas Distribuídos Tradicionais

De acordo com as figuras, percebe-se que em comum os Sistemas Distribuídos Tradicionais (Figura 1) e os Sistemas de Código Móvel (Figura 2) compartilham apenas as camadas *COS - Core Operating System* e *NOS - Network Operating System* (FUGGETTA; PICCO; VIGNA, 1998). Sendo a camada *COS* a mais próxima ao hardware, e cuja a responsabilidade está em prover as funcionalidades básicas do sistema operacional, tais como: sistema de arquivos, gerenciamento de memória e suporte dos processos. Vale ressaltar que não há suporte para comunicação ou distribuição nesta camada.



CE - Ambiente Computacional; NOS - Sistema Operacional de Rede;
COS - Núcleo do Sistema Operacional; HOST - Máquina pertencente a rede.

Figura 2: Sistemas de Código Móvel

Cabe à camada *NOS* oferecer suporte aos serviços de comunicação não transparente. As aplicações usando a camada *NOS* endereçam explicitamente a máquina de destino pela comunicação. Conceitualmente podemos dizer que a camada *NOS* utiliza os serviços oferecidos pela camada *COS*, como no caso do gerenciamento de memória (FUGGETTA; PICCO; VIGNA, 1998).

A principal diferença entre os Sistemas Distribuídos Tradicionais e Sistemas de Código Móvel está na terceira camada, nos sistemas tradicionais a camada TDS[4], é responsável tanto pela transparência da rede como pela implementação da plataforma onde os componentes, mesmo alocados nos diferentes nós de uma rede, são percebidos como local. Nas tecnologias que suportam código móvel encontramos o CE[5], ou ambiente computacional, uma camada acima do *NOS* de cada máquina da rede. Diferentemente do TDS, o CE possui a capacidade de armazenar a identidade da máquina onde está sendo executado, o que posteriormente irá permitir que as aplicações realoquem dinamicamente seus componentes nos diferentes nós da rede .

Em um TDS, os usuários não precisam conhecer a estrutura da rede, uma vez que ao ser invocado, não há indicação sobre o nó da rede que realmente irá prover aquele serviço, ou mesmo se há uma rede presente. Em um Sistema de Código Móvel por outro lado, esses usuários terão uma perspectiva diferente, tendo em vista que a estrutura básica da rede não será mais escondida.

[4]Do inglês: *True Distributed System*
[5]Do inglês: *Computational Environment*

Os componentes hospedados por um CE podem ser classificados em (FUGGETTA; PICCO; VIGNA, 1998):

- Unidades de Execução - *Executing Units* (EU), representam fluxos seqüenciais de computação, por exemplo, *threads* individuais em processo multi-*threaded*.
- Recursos - *Resources*, representam entidades que podem ser compartilhadas entre múltiplas EU's, por exemplo, um arquivo em um sistema de arquivos.

2.2.1 Mecanismos de Mobilidade

Os mecanismos de mobilidade de código dos Sistemas de Código Móvel, apresentados na Figura 3, podem ser de duas formas (FUGGETTA; PICCO; VIGNA, 1998; TANENBAUM; STEEN, 2002):

Mobilidade Forte - habilidade que um MCS (item 2.1 - p. 8), também conhecido como *strong MCS*, em permitir a migração tanto do código como do estado de execução de uma EU para um diferente CE. São suportados dois tipos de mecanismos: Migração e Clonagem Remota, sendo que no primeiro ocorre a suspensão da EU, a transmissão desta à CE de destino e então é retomada a execução, no segundo mecanismo, é criada uma cópia da unidade de execução no ambiente computacional de destino.

Mobilidade Fraca - habilidade de um MCS (item 2.1 - p. 8), também conhecido como *weak MCS*, de permitir a transferência de código através de diferentes ambientes computacionais. Essa transferência pode ser classificada por exemplo, de acordo com a direção da transferência de código, uma vez que a unidade de execução pode tanto requerer dinamicamente (*fetch*) o código necessário, quanto remeter (*ship*) este código a outro CE.

As tecnologias atualmente disponíveis diferem na forma de implementação dos mecanismos de suporte à mobilidade de código. As Tabelas 1 e 2 apresentam as principais tecnologias relacionadas à mobilidade de código e qual é o mecanismo de suporte à migração implementado.

Vale ressaltar que grande destaque tem sido dado à linguagem Java, desenvolvida pela *SUN Microsystems*. Essa linguagem tem despertado grande interesse, uma vez que

Tecnologia	Origem
Facile	Desenvolvida pelo *European Computer Industry Research Center* em *Münich*, é uma linguagem funcional que estende a linguagem *Standard ML* com primitivas para distribuição, concorrência e comunicação. As unidades de execução são implementadas como *threads* que executam em ambientes computacionais Facile, chamados de nós.
Linguagem Java	Desenvolvida pela *Sun Microsystems* tem despertado mais atenção, devido à facilidade de programação e a portabilidade. Ao compilar um arquivo .java é gerada uma classe que pode ser transferida, através de protocolos, de um cliente para um servidor. A classe compilada está em formato de *bytecode*, que é interpretado pela máquina virtual; Portanto, basta a plataforma possuir a JVM para portar classes Java sem a necessidade de alteração.
Java *Aglets*	A API Java *Aglets*, desenvolvida pelo Laboratório de Pesquisa de Tóquio da IBM, Japão, estende a linguagem Java com suporte à mobilidade fraca. *Aglets* (unidades de execução) são *threads* em um interpretador Java que constituem neste cenário o ambiente computacional.
M0	Implementada pela Universidade de Genebra, Suiça, é uma linguagem interpretada baseada em pilha e que implementa o conceito de mensageiros. Mensageiros (unidades de execução) são seqüências de instruções que são transmitidas entre plataformas (ambientes computacionais) e executados incondicionalmente conforme recebimento.
Mole	Desenvolvida pela Universidade de *Stuttgart*, é uma API Java, onde agentes Mole são objetos Java que são executados como objetos do tipo *threads* da máquina virtual Java, que é abstraído dentro de um lugar, o ambiente computacional Mole.
Obliq	Desenvolvida pela DEC, é um tipo diferente uma vez que é baseada em objetos e lexicamente extensa, sendo também uma linguagem interpretada. Uma *thread* (unidade de execução), pode requisitar a execução de um procedimento em uma máquina de execução remota. O código para tal procedimento é enviado para a máquina de destino e executado lá por uma nova unidade de execução.
TACOMA	Na linguagem Tcl, unidades de execução, chamadas agentes, são implementadas como processos Unix executando o interpretador Tcl.

Tabela 1: Tecnologias que oferecem suporte à Mobilidade Fraca (FRAGA; WANGHAM, 2001).

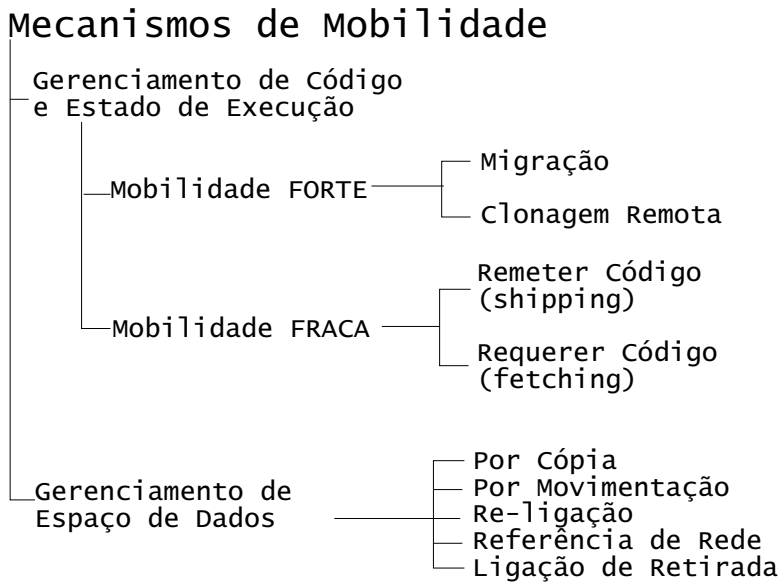


Figura 3: Mecanismos de Mobilidade de Código(FUGGETTA; PICCO; VIGNA, 1998)

desde sua concepção foi projetada para ser uma linguagem orientada a objetos de propósito geral, portável, clara e fácil de ser aprendida.

Tecnologia	Origem
Tcl	Desenvolvida pela Universidade de *Darthmouth*, provê um interpretador Tcl estendido. Unidades de execução executam em universos de endereços isolados, elas podem compartilhar somente recursos fornecidos pelo sistema operacional básico, como arquivos. A abstração do ambiente computacional é implementada pelo sistema operacional e pela linguagem de suporte em tempo de execução.
ARA	Desenvolvido na Universidade de *Kaiserslautern*, é uma multi-linguagem sendo que as unidades de execução, chamadas de agentes. São gerenciadas pelo núcleo do sistema, independente da linguagem, mais há interpretadores para as linguagens suportadas - C, C++ e Tcl. O núcleo e os interpretadores constituem o ambiente computacional, cujo os serviços são disponibiizados aos agentes através da abstração de lugar (*place*)
SUMATRA	Desenvolvida pela *Universidade de Maryland*, é uma extensão da linguagem Java, desenhada expressamente para suporte à implementação de programas móveis *resource-aware* - programas que são capazes de adaptem-se a mudanças de recurso.
Telescript	Desenvolvida pela General Magic, é uma linguagem orientada a objetos concebida para o desenvolvimento de aplicações distribuídas. A segurança tem sido um dos principais fatores de condução do projeto dessa linguagem.

Tabela 2: Tecnologias que oferecem suporte à Mobilidade Forte. (FRAGA; WANGHAM, 2001)

O compilador Java traduz os programas fonte (`.java`) em um formato intermediário chamado *Java bytecode* (`.class`), que é posteriormente interpretado pela JVM - *Java Virtual Machine* - a implementação do CE. A linguagem Java provê um mecanismo programável, o *class loader*, que permite consultar e vincular dinamicamente classes que estão executando em uma JVM. O *class loader* é invocado pela JVM em tempo de execução toda vez que o código contém um nome de classe não resolvido.

Por essa razão a linguagem Java, oferece suporte à mobilidade fraca por meio de mecanismos assíncronos que possibilitam a transferência de fragmentos de código (*fetching*), tanto por execução imediata como adiada.(FUGGETTA; PICCO; VIGNA, 1998)

Um dos fatores de sucesso da linguagem Java está justamente na sua capacidade de integração com a WWW. Atualmente, grande parte dos navegadores *Web* incluem a máquina virtual Java, sendo que *applets* Java, podem ser executados no navegador *Web* a partir de páginas *Hyper Text Markup Language* - HTML. Partindo da perspectiva de que a combinação WWW e JVM constitui por si só em uma tecnologia, o fato da JVM estar escondida, faz com que seus mecanismos sejam usados para prover uma camada de alto nível. O que permite que os navegadores *Web* atuem como CEs e *applets* Java desempenhariam o papel de EUs em execução concorrente neste ambiente. Nesse contexto, a transferência de *applets* Java pode ser registrada com um mecanismo provido pelo navegador para suportar *fetching* de código.(FUGGETTA; PICCO; VIGNA, 1998)

2.3 Paradigmas de Projeto

A tecnologia de mobilidade de código é apenas um dos ingredientes necessários para construir um software. O processo de desenvolvimento de software é um processo complexo em que vários fatores devem ser considerados: organização, tecnologia e metodologia. Paradigmas de Projeto dizem respeito a arquiteturas de software que compartilham características similares, ou seja, as abstrações de arquiteturas e estruturas que podem ser instanciadas no projeto de arquitetura de um software. Há separação entre a tecnologia usada e o processo de desenvolvimento - que são tratados conceitualmente como entidades distintas.

Por não oferecem a flexibilidade e a escalabilidade desejadas, as abordagens tradicionalmente utilizadas no projeto de software não tem sido suficientes quando aplicadas

ao projeto de aplicações distribuídas para ambientes de larga escala, como a Internet, que exploram mobilidade de código e reconfiguração dinâmica dos componentes de software. Nesses casos, conceitos como: localidade, distribuição de componentes entre localidades e migração de componentes para diferentes localidades, precisam ser tratados explicitamente durante todos os estágios do projeto, uma vez que a interação entre componentes em uma mesma máquina é notadamente diferente do caso onde os componentes residem em diferentes máquinas de uma rede. Interações entre componentes residentes em um mesmo nó são consideradas menos exigentes do que interações que ocorrem entre componentes localizados em nós distintos de uma rede.

Fuggeta *et al.* (FUGGETTA; PICCO; VIGNA, 1998), cita 3 conceitos arquiteturais básicos, que nada mais são que abstrações das entidades que constituem um sistema de software:

- **Componentes** - são as partes da arquitetura do software. Eles são divididos entre **componentes de código**, que encapsulam o *know-how* para executar uma computação em particular, **componentes de recurso**, que representam dados ou equipamentos usados durante a computação, e **componentes computacionais** que representam o ambiente onde irá ocorrer a computação.
- **Interações** - são eventos que envolvem dois ou mais componentes, por exemplo, uma mensagem trocada entre dois componentes computacionais.
- **Sítios** - também chamados de nós da rede, hospedam os componentes e oferecem suporte à execução de componentes computacionais. Um sítio representa intuitivamente a noção de local.

Paradigmas de Projeto (FUGGETTA; PICCO; VIGNA, 1998; PICCO, 2001a) são descritos em termos de Padrões de Interações que definem a realocação e a coordenação entre componentes necessários a execução de um serviço.

A seguir são descritos os principais paradigmas de projetos aplicados à mobilidade de código:

2.3.1 Cliente-Servidor

O paradigma cliente-servidor - CS[6], além de bem conhecido pela comunidade é amplamente usado. Nele, um componente computacional B (o servidor) oferece um conjunto de serviços alocados em S_b. Os recursos e o conhecimento necessário à execução dos serviços também estão alocados em S_b. O componente cliente A, localizado em S_a, requisita a execução de serviços por meio de uma interação com o componente servidor B.

Como resposta, B executa o serviço requisitado por meio da aplicação do conhecimento correspondente e da utilização dos recursos envolvidos co-localizados em B. Em geral, o serviço produz algum resultado que será devolvido ao cliente por meio de interações adicionais.

2.3.2 Avaliação Remota

No paradigma de Avaliação Remota - REV[7], o componente A possui o conhecimento necessário para executar o serviço, mas ele não possui os recursos requeridos, que aparentemente estão alocados no sítio remoto S_b. Dessa forma, A envia o conhecimento ligado àquele serviço para o componente computacional B localizado no nó remoto. B executa o código usando os recursos disponíveis e devolve os resultados para A.

2.3.3 Código sob Demanda

No paradigma de Código sob Demanda - COD[8], o componente A está apto a fazer uso dos recursos necessários, que estão co-localizados em S_a. Entretanto, não há informação sobre como manipular esses recursos, o que faz com que A interaja com o componente B solicitando o conhecimento presente em B. Outra interação acontece quando B envia esse conhecimento a A que poderá então executá-lo.

[6]Do inglês: *Client-Server*
[7]Do inglês: *Remote Evaluation*
[8]Do inglês: *Code on Demand*

2.3.4 Agentes Móveis

No Paradigma de Agentes Móveis - MA[9], o conhecimento do serviço é de propriedade de A, que é inicialmente hospedado por S_a, mas alguns dos recursos necessários estão alocados em S_b. Isso faz com que A migre para S_b transportando o conhecimento e possivelmente alguns resultados intermediários.

Após a transferência, o serviço é realizado por A usando os recursos disponíveis em S_b. Os agentes móveis diferem dos demais paradigmas devido às interações associadas envolverem a mobilidade de um componente computacional já existente, enquanto que nos paradigmas REV e COD o enfoque está na transferência do código entre componentes.

2.4 Domínio de Aplicações

Quando comparadas às aplicações cliente/servidor tradicionais, a programação de aplicações explorando a idéia de mobilidade de código ainda pode ser considerada como relegada a um nicho específico, o que em parte se deve à imaturidade da tecnologia e à falta de metodologias para desenvolvimento de aplicações. Entretanto, vale ressaltar, que o interesse não está na tecnologia de mobilidade de código, mas sim, nos benefícios possíveis de serem alcançados por essa nova forma de construção de aplicações distribuídas.

As vantagens esperadas com a introdução dos códigos móveis em aplicações distribuídas são o principal apelo em vários domínios de aplicação. Contudo, o entendimento da tecnologia passa pela clara diferenciação entre uma aplicação (por exemplo, um sistema para controlar um telescópio remoto) e o paradigma usado para projetá-lo (o paradigma da Avaliação Remota para identificação dos módulos de controle que são enviados ao telescópio remoto) ou a tecnologia usada na sua implementação (por exemplo, *Java Aglets* (FUGGETTA; PICCO; VIGNA, 1998) - propostos pela IBM, Japão).

Os conceitos e as tecnologias de mobilidade de código englobam também noções de **autonomia** dos componentes da aplicação, que é uma propriedade útil para aplicações que usam uma infra-estrutura de comunicação heterogênea onde os nós de uma rede podem ser conectados por uma variedade de vínculos físicos com desempenhos diferentes. A autonomia dos componentes de uma aplicação promove, como efeito colateral, a melhoria

[9]Do inglês: *Mobile Agent*

da tolerância a falha. Considera-se também o gerenciamento flexível dos dados e encapsulamento do protocolo como vantagens atribuídas à introdução da mobilidade de código no desenvolvimento de aplicações distribuídas.

Fuggetta *et al.* (FUGGETTA; PICCO; VIGNA, 1998), destaca alguns domínios de aplicação que utilizam, ou que podem vir a ser beneficiados, pela exploração dos códigos móveis:

- Consulta Distribuída de Informações;
- Documentos Ativos;
- Configuração e Controle de Equipamentos Remotos;
- Serviços Avançados de Telecomunicação, e
- Comércio Eletrônico.

2.5 Implicações sobre Segurança

A mobilidade de código tem causado profundas mudanças no modo como as aplicações distribuídas, em especial aquelas de larga escala, são concebidas e construídas. Contudo, na mesma proporção em que significantes oportunidades são oferecidas, cresce também a preocupação com questões relacionadas à segurança dessas aplicações.

No cerne dessa discussão, podemos colocar as implicações e os riscos associados à transferência e execução de um código criado por outrem, sendo que, de modo em geral busca-se assegurar a prevenção ao acesso e à manipulação ilegítima da informação, ou ainda, de evitar a interferência indevida na sua operação normal.

Contudo, vale ressaltar que este tipo de sistema apresenta diversos fatores que o torna muito peculiar, dentre os quais destacamos (FRAGA; WANGHAM, 2001):

- **Arquitetura** - tais sistemas por natureza, envolvem o uso de diferentes nós em uma rede de computadores. Sendo que podemos assumir que existe uma Plataforma Raiz e uma ou mais Plataforma de Agentes, pelas quais esses elementos do sistema irão migrar e interagir com. Assim, o dono do agente e operador da plataforma de agentes podem vir a ser pessoas diferentes.

- **Vulnerabilidades** - uma plataforma para trabalho com agentes móveis em geral agrega, ou faz uso de um conjunto de elementos heterogêneos como: ambientes computacionais, redes de computadores, políticas de segurança, entre outros. Dessa forma, os desenvolvedores também devem lidar com as questões ligadas as vulnerabilidades desses elementos.

- **Meio de Comunicação** - em geral, tais sistemas se valem dos recursos oferecidos pela Internet para realizar a sua migração, o que implica no aumento das vulnerabilidades do sistema, uma vez que este fica sujeito a ataques por parte de qualquer elemento do sistema (plataforma e agente) ou mesmo externo.

Fraga e Wangham (FRAGA; WANGHAM, 2001) também destacam que a mobilidade de código envolve o desenvolvimento de aplicações e serviços mais flexíveis, dinâmicos e customizados, sendo que em geral envolve um software que viaja através de uma rede heterogênea, atravessando domínios de segurança e sendo executado automaticamente no seu destino.

Segundo Gritzalis e Iliadis (GRITZALIS; ILIADIS, August 1998), segurança é a prática pela qual organizações e indivíduos protegem suas propriedades físicas e intelectuais de todas as formas de ataque e saque de informações. Em geral a segurança em sistemas de informática, visa oferecer a capacidade de assegurar a prevenção ao acesso e à manipulação ilegítima da informação, ou ainda, de evitar a interferência indevida na sua operação normal. No caso dos Agentes Móveis, vale ressaltar que os problemas relacionados à segurança prejudicam a ampla aceitação das abordagens propostas, as Tabelas 3 e 4 apresenta, respectivamente, um breve histórico com as preocupações relacionadas à segurança e uma visão geral das categorias de ameaças ao qual tais sistemas estão expostos (FRAGA; WANGHAM, 2001).

Apesar de não se tratar de um problema totalmente desconhecido, a crescente utilização da WWW por um número cada vez maior de pessoas, trouxe novas perspectivas e novos significados às questões relacionadas à segurança das aplicações distribuídas. Visto que com o aumento do uso das redes de computadores, do suporte oferecido aos novos serviços e a necessidade de diferentes níveis de acesso, também aumentaram os ataques oriundos de agentes externos ao sistema, que visam a explorar as vulnerabilidades.

Quando do início das chamadas Aplicações Distribuídas Tradicionais, certo consenso já existia a respeito dos problemas vinculados à execução do código de uma pes-

soa, na maior parte das vezes, desconhecida. Nesta época, cabia ao usuário do sistema conscientemente decidir se deveria ou não transferir e instalar os executáveis de uma determinada origem (MCGRAW; FELTEN, 1998).

Uma das principais dificuldades enfrentadas nos dias atuais reside no fato de que os usuários em certas ocasições podem não ter a percepção de que estão requisitando, ou mesmo, executando um código móvel em seu computador (MCGRAW; FELTEN, 1998).

Período	Comentários
No início da Internet	Consenso geral sobre as implicações de transferir e executar binários.
Metade da década de 80	Aumento do número de software do tipo *freeware* e *shareware* disponíveis para transferência.
Final da década de 80	Utilização do recurso de *checksumming* - computação simples, realizada em um código como forma de imprimir, por meio de uma função *hash one-way* uma identificação a esse código - como forma de garantir que ao final da transferência o código transferido era o mesmo originalmente distribuído pelo autor.
Anos 90	1992 - No início as páginas HTML eram estáticas, o que foi alterado apenas com o aparecimento dos sistemas de código móvel, tais como Java e JavaScript que possibilitaram aos servidores *WWW* fornecerem programas na forma de conteúdo.

Tabela 3: Breve Histórico das Preocupações relacionadas com Segurança

Vale ressaltar, conforme a Tabela 3, que o que antes era uma decisão consciente por parte dos usuários e que exigia vários passos em um processo tecnológico, agora exige apenas um simples clicar com o mouse em uma ligação WWW.

Com relação às ameaças relacionadas ao uso de Agentes Móveis, Fraga e Wangham (FRAGA; WANGHAM, 2001) propõem uma outra categorização conforme apresenta a Tabela 4.

Em linhas gerais, o uso da mobilidade de código exige a proteção das plataformas da chamada computação maliciosa, e vice-versa, o estabelecimento de meios para que a comunicação entre a plataforma e os agentes móveis garanta a integridade e a confidencialidade do estado e do código do agente móvel (FRAGA; WANGHAM, 2001).

A seguir são destacadas algumas formas de proteção que deveriam ser garantidas, ao se trabalhar com ambientes de Agentes Móveis:

Categoria de Ameaças	Comentários
Agentes *vs.* Plataforma de Agentes	Agentes Móveis podem atacar uma Plataforma de Agentes, visando promover a negação de serviços (DoS, do inglês:*Denial of Service*), além de roubo e modificação dos dados armazenados na plataforma de agentes.
Plataforma de Agentes *vs.* Agentes	Com intuito de adquirir informações, ou mesmo violar um sistema, uma Plataforma de Agentes pode realizar ataques a um Agente Móveis por meio de mascaramento, ou de intromissão, uma plataforma de agentes maliciosa pode tentar se passar por um nó válido dentro da rede.
Agentes *vs.* Agentes	Em função das mesmas motivações envolvidas no ataque de uma Plataforma de Agentes a um Agente, pode também ocorrer o ataque entre Agentes, que por meio de repudiação e mascaramento, buscam obter informações.
Elementos Externos *vs.* Sistema (Plataforma e Agentes)	Visando o roubo de dados, outras entidades externas ao sistema também podem realizar ataques a sistemas de Agentes Móveis, através de mascaramento ou mesmo da análise de dados entre Plataformas de Agentes.

Tabela 4: Categorias de Ataques - Agentes Móvies.

- A Plataforma de Agentes deve ser protegida da chamada computação maliciosa e vice-versa.
- Os acessos aos recursos locais devem ser controlados como forma de proteger os nós de uma rede, ou as Plataformas de Agentes.
- Os Agentes devem ser autenticados.
- A comunicação entre a Plataforma deve garantir a integridade e a confidencialidade do estado e do código do agente.
- O agente deve se proteger mutuamente.

O interesse nas questões relacionadas à segurança dos Agentes Móveis está ligada ao fato de que embora nem todos os componentes sejam móveis, os agentes móveis serão uma parte essencial de muitos sistemas distribuídos, sendo que atualmente já há um número crescente de aplicações nas mais diversas áreas. Contudo, ressalta-se que a tecnologia ainda está amadurecendo, e ainda existem obstáculos para a ampla adoção desta tecnologia em função de questões como a segurança, robustez e padronização. Wangham (FRAGA; WANGHAM, 2001) apresenta uma distinção entre as plataformas disponíveis classificando-as em comerciais e acadêmicas, conforme mostram as Tabelas 5 e 6.

Plataforma	Endereço
Aglets	http://www.trl.ibm.co.jp/aglets.
Concórdia	http://www.meitca.com/HSL/Projects/Concordia.
Grasshopper	http://www.ikv.de/products/grasshopper.
Voyager	http://www.objectspace.com/voyager1.htm.

Tabela 5: Plataformas Comerciais

Plataforma	Endereço
Mole	http://www.informatik.uni-stuttgart.de/ipvr/vs/project/mole.htm.
MOA	http://www.camb.opengroup.org/RI/java/moa/index.htm.
Agent Tcl	http://www.cd.dartmouth.edu/ agent.
SOMA	http://www-lia.deis.unibo.it/Software/SOMA.
Ajanta	http://www.cs.umn.edu/Ajanta.

Tabela 6: Plataformas Acadêmicas

Destaca-se que as algumas plataformas possuem seu foco no uso de alguma linguagem de programação, enquanto que as demais centram seus recursos nas primitivas

de programação. Dentre as chamadas *plataformas acadêmicas* apresentamos um breve resumo das características suportadas:

- **Mole**
  - Comunicação Segura - Não suporta.
  - Proteção da Plataforma - Modelo sandbox (implementado simples), gerenciamento de recursos.
  - Proteção do Agente - F. Hohl - caixa preta limitada, focada no estado.
  - Outro - Elementos Externos- Controle do Agente, tolerância. (*exactly-one*)
- **MOA**
  - Comunicação Segura - Não suporta. Há planos para assinatura dos JAR.
  - Proteção da Plataforma - Security Manager não está implementado.
  - Proteção do Agente - Não suporta.
  - Outros - Elementos Externos- Controle de agentes. Planos: gerenciamento de recurso (negação de serviço)
- **Agent Tcl**
  - Comunicação Segura - PGP para autenticação e cifragem.
  - Proteção da Plataforma - Safe-Tcl
  - Proteção do Agente - Não suporta
  - Outros - Elementos Externos- Não suporta (controle, *check point*).
- **SOMA**
  - Comunicação Segura - Não suporta
  - Proteção da Plataforma - JDK 1.2, Autenticação (PKI) e autorização (RBAC), atributos estáticos e dinâmicos.
  - Proteção do Agente - TTP (3a parte confiável) e MH (detecção = rastro ex-integridade do estado)
  - Outros - Elementos Externos- Controle de Agentes
- **Ajanta**

- Comunicação Segura - Transferências cifradas e autenticadas (elGamal e DAS)
- Proteção da Plataforma - Acesso aos recursos baseado em capabilities. Autorização (dono do agente).
- Proteção do Agente - Mecanismos para detectar modificações no código e estado.
- Outros - Elementos Externos- Controle de agentes (retract, recall e terminate)

Rubin e Geer Jr. (RUBIN; JR, 1998) propõem quatro abordagens voltadas ao tratamento das questões de segurança em mobilidade de código: *sandbox*, *code signing*, *firewalling* e *proof-carrying code*. Vale ressaltar que, enquanto a primeira abordagem limita os privilégios dos executáveis a um pequeno conjunto de operações, o chamado modelo ***sandbox***. A segunda procura meios de assegurar que o código a ser executado é confiável, o que é conhecido por ***code signing***.

Já a terceira abordagem, ***firewalling***, examina os executáveis quando eles entram em um domínio confiável e decide se e como eles serão executados no cliente. Por fim, a quarta abordagem, chamada de ***proof-carrying code*** está atualmente limitada ao uso de programas em linguagem de montagem escrito por desenvolvedores da abordagem. Nesta técnica, o código móvel é carregado com uma demonstração de que satisfaz certas propriedades.

Destaca-se ainda a existência de uma outra abordagem, híbrida, oriunda da combinação entre duas técnicas, *sandbox* e *code signing*, implementada na versão 1.2 do JDK[10] (J2SDK) e no navegador da *Netscape Communicator*[11].

As sessões seguintes comentam as abordagens citadas (RUBIN; JR, 1998).

2.5.1 O Modelo *Sandbox*

De acordo com Rubin e Geer (RUBIN; JR, 1998), a idéia implementada pelo modelo *sandbox* é conter a execução do código móvel de tal forma que não haja danos ao ambiente de execução. O que é feito por meio da restrição de acesso ao sistema de arquivos e pela limitação na habilidade de criar conexões com a rede.

[10] *Java Development Kit*, kit de desenvolvimento da linguagem Java, disponibilizado pela *Sun Microsystems*, através do endereço: http://java.sun.com

[11] Disponível no endereço: http://www.netscape.com

A mais difundida das implementações do modelo *sandbox* é o interpretador da linguagem Java, presente nos navegadores Internet. O Modelo de Referência de Segurança para o JDK[12] versão 1.0.2, oferece uma excelente visão dos requisitos fundamentais de segurança para o ambiente Java.

O principal problema relacionado ao *sandbox* Java é que qualquer erro em algum componente de segurança pode levar a uma violação das politicas de segurança. Os riscos são exarcebados pela complexidade das interações entre os componentes.

2.5.2 *Code Signing*

Nesta abordagem o servidor gerencia uma lista de entidades em que, ou quem, ele confia. Ao receber um código verifica-se se ele foi assinado por uma das entidades pertecentes a essa lista, e em caso afirmativo o ambiente irá executar o código, frequentemente com todos os privilégios de usuário (RUBIN; JR, 1998).

O modelo de confiança para códigos *code signing* parte da premissa de que é possível distinguir autores de código móvel confiável dos não-confiáveis, e que autores confiável são incorruptíveis.

2.5.3 *Firewalling*

Envolve seletivamente a escolha entre executar ou não um programa no momento em que esse código chega ao domínio do cliente. Uma situação que exemplifica essa situação pode ser considerada quando uma organização possui um *Web proxy* ou um *Firewall*, e o usa para tentar identificar *applets* Java, examinando-os e procurando decidir se eles são úteis ou não ao cliente.

Segundo Rubin e Geer (RUBIN; JR, 1998), pesquisas mostram que a tarefa de definir quais aplicativos podem ou não ser executados não é tão simples. A técnica de *firewalling* assume a existência de alguma forma de identificação para *applets*.

[12] *Security Reference Model for JDK 1.0.2, Sun Microsystems*

2.5.4 *Proof-Carrying Code*

É uma técnica para verificação estática do código, que visa garantir a não violação de nenhuma política de segurança. Por exemplo, para alguns programas é possivel construir uma prova de que não haverá qualquer situação de *buffer overflow*(RUBIN; JR, 1998).

Entretanto há propriedades relacionadas com o fluxo de informações e a confidencialidade que não podem ser provadas desta maneira. O modelo de confiança adotado nessa técnica pode ainda ser alterado. Até o presente momento, o projeto e a implementação do verificador são considerados confiáveis mas o código móvel é universalmente não-confiável (RUBIN; JR, 1998).

Vale ressaltar que há uma nova abordagem híbrida que se propõe a fundir os benefícios das abordagens propostas pelo modelo *sandbox* e *code signing*. O modelo de confiança utilizado adota como premissa que todos os códigos são não-confiáveis. Exceto aqueles códigos oriundos de fornecedores confiáveis, previamente identificados, que serão considerados incorruptiveis (RUBIN; JR, 1998).

Atualmente, diversas abordagens relacionadas às questões sobre segurança em mobilidade de código estão em desenvolvimento. As diferentes implementações de código móvel compartilham riscos de segurança similares, contudo, cada sistema possui uma maneira própria de tratá-los.

Segundo McGraw e Felten (MCGRAW; FELTEN, 1998), devido às suas características, a segurança na linguagem Java merece destaque nesse cenário. Porém, assim como em outras formas de mobilidade de código, seu mecanismo de segurança não é trivial - fazendo de sua implementação muito mais uma arte do que uma ciência.

Criar e manter sistemas seguros requer rigorosa garantia do software e a habilidade de gerenciar riscos. Código Móvel torna a tarefa de assegurar um sistema mais complicada, uma vez que os benefícios provenientes do uso de código móvel estará frequentemente associado a algum risco adicional. melhores resultados podiam ser alcançados por meio da combinação das diferentes abordagens mencionadas (MCGRAW; FELTEN, 1998; RUBIN; JR, 1998).

2.6 O Ambiente μCode

Como reflexo do crescente interesse pelos benefícios advindos da exploração da migração de código entre as máquinas de uma rede, várias abstrações e mecanismos foram propostos. Entretanto, vale ressaltar que em sua grande maioria essas abordagens foram caracterizadas pela pouca flexibilidade e por exigirem grande processamento (PICCO, 1998).

O μCode emerge neste contexto com o intuito de ser uma alternativa às abordagens voltadas à exploração da mobilidade de código, desde sua concepção, e posterior projeto, objetivou-se oferecer um ambiente que apresentasse a flexibilidade e a extensibilidade necessárias às aplicações distribuídas voltadas a ambientes de larga escala como a Internet, e que não sobrecarregasse demasiadamente o processamento (PICCO, 1998).

Segundo seu criador (PICCO, 1998), um dos principais diferenciais do ambiente para mobilidade de código μCode quando comparado às outras tecnologias afins, está na capacidade de não ser apenas um sistema de agentes móveis - *mobile agent system* (PICCO, 1998). O que significa dizer, que seu principal enfoque não são apenas os agentes móveis, outros paradigmas de mobilidade de código são também contemplados.

O μCode, implementado na forma de uma pequena API escrita em linguagem Java, foi idealizado com foco nos mecanismos que habilitam a mobilidade de código. Dessa forma, oferece aos programadores a possibilidade de tanto fazerem uso das primitivas de migração providas pelo μCode, como implementar novas primitivas de acordo com a própria noção de migração de código e que melhor atendam às eventuais necessidades desses programadores (PICCO, 2001b).

De acordo com os critérios definidos no projeto do ambiente, este não tem por objetivo ser um ambiente completo voltados aos agentes móveis. O μCode procura oferecer boas abstrações dos mecanismos de mobilidade de código de forma a tornar possível realizar a mobilidade de código.

Dessa forma, podemos assumir que sua estrutura é análoga aos sistemas operacionais modernos. O μCode constitui o núcleo do sistema, oferecendo um mecanismo pequeno e eficiente para mobilidade de código, e ao seu redor, podem ser desenvolvidos um conjunto de módulos que ofereçam funcionalidades especializadas, e opcionais (PICCO, 2001b).

Com base na descrição do ambiente μCODE (PICCO, 2001b, 1998), destacamos as seguintes características:

- Trabalha com linguagem Java, versão 1.1 ou posterior;
- Baixos requisitos - a atual implementação, além de exigir poucos recursos de processamento (*lightweight*), ocupa por volta de 40 KBytes de *bytecode*, sendo que o pacote núcleo do ambiente possui menos de 18 Kbytes - armazenados em um arquivo do tipo jar;
- Projeto aberto, empenho pela modularidade e flexibilidade;
- Suporte a qualquer estratégia de realocação de classe - desde Vínculo Remoto Dinâmico à transferência (*shipping*) de uma classe inteira;
- Suporte a compressão GZIP na comunicação entre *hosts*.

A Tabela 7 apresenta os princípios que nortearam o projeto do ambiente(PICCO, 1998):

Vale ressaltar, que o ambiente μCODE é um projeto *open source*, com o uso de arquivos fontes e binários regulados pelo *GNU Lesser General Public License*.

2.6.1 Estrutura do Ambiente μCode

De acordo com sua documentação (PICCO, 1998), as operações básicas suportadas pelo μCODE permitem tanto a criação e cópia de objetos do tipo *thread* - um servidor remoto ***μServer*** - bem como a realocação de classes entre esses tipos de servidores.

Um servidor μServer é uma abstração do suporte em tempo de execução e representa um ambiente computacional (CE) para *threads* móveis. A respeito da migração, os objetos, inclusive *threads*, mantêm o estado dos seus dados porém perdem seu estado de execução. Assim, dentre os mecanismos descritos na taxonomia proposta por Fuggetta *et al.*(FUGGETTA; PICCO; VIGNA, 1998), somente mobilidade fraca é suportada (item 2.2.1 - p. 11). O μCODE possibilita tanto *remeter* (*code shipping*) como *requerer* (*code fetching*) fragmentos de código ou códigos *stand-alone*, seja por invocação síncrona ou assíncrona, bem como por execução de código móvel imediata ou adiada (PICCO, 1998).

Princípio	Comentário
Flexibilidade	A flexibilidade apresenta-se em diferentes facetas: • *Controle sobre realocação de código* - para o mesmo código móvel, diferentes estratégias devem estar disponíveis de modo a atender diferentes situações. • *Acesso a diferentes níveis de abstração* - Embora construções de alto-nível seja providas pelo ambiente, construções de nível menor devem ser acessíveis quando alguma funcionalidade em particular, ou customizada, seja necessária. • *Controle sobre o suporte em tempo de execução* - O suporte em tempo de execução pode ser controlado diretamente pelos programas, e não por programas externos.
Minimalidade	O conjunto de conceitos e mecanismos deve empenhar-se por minimalidade, desde modo, favorecendo a composabilidade e a facilidade de entendimento.
Não-Invasivo	Mobilidade de Código é mais uma opção aos programadores, cujo o esforço para lidar com mobilidade deve ser minimizado e a liberdade de projeto maximizada.
Extensibilidade	μCODE é uma "fina" camada que oferece o núcleo das funcionalidades sob a JVM - *Java Virtual Machine*. Seu projeto deve ser aberto, cada uma das suas primitivas básicas pode ser composta em abstrações de alto-nível, ou suportes especializados em tempo de execução podem ser construídos.
Portabilidade	O ambiente deve ser plenamente compatível com a linguagem Java, de modo a operar em todas as plataformas suportadas por esta linguagem.

Tabela 7: Princípios de Projeto - Ambiente μCODE

No μCODE, a unidade de migração é chamada de ***group***, sendo essa unidade um container que pode vir a ser arbitrariamente preenchido com classes e objetos, e remetidas a um destino. As classes e os objetos armazenados neste repositório, não necessitam pertencer somente à mesma unidade de execução, o programador pode optar, por exemplo, por inserir em um grupo somente classes que realmente serão necessárias em um destino, e explorar o Vínculo Remoto Dinâmico[13] para transferir as classes adicionais a partir de um outro nó específico (PICCO, 1998, 2001b).

O destino de um *group* é um ***μServer***, o que segundo Picco (PICCO, 2001b) é uma abstração do suporte em tempo de execução do μCODE, dando suporte aos mecanismos básicos para realocação de código e estado. Uma vez no destino, o conjunto de classes e objetos deve ser extraído a partir do *group*, tarefa que cabe ao chamado ***group handler*** - uma classe especificada pelo programador em tempo de criação do *group*, que é instanciada e cujas operações são automaticamente invocadas no destino (PICCO, 2001b).

Enquanto o *group* é reconstruído no ***μServer***, o sistema precisará alocar as classes e torná-las disponíveis para subseqüentes execuções de um *group handler*. Durante o processo de extração, ao serem extraídas, as classes são colocadas nos chamados *namespace*'s.

Deste ponto, criam-se os conceitos de ***class space*** privado e compartilhado, permitindo assim que classes remetidas em um mesmo *group* sejam alocadas em um *class space* privado. Posteriormente, essas mesmas classes podem ser *publicadas* em um *class space* compartilhado, associado a um *μServer* o que as tornariam visíveis a todas as *threads* que executam no servidor.(PICCO, 2001b)

A distribuição do μCODE é composta por três pacotes (PICCO, 2001b):

- `mucode` - Contém o núcleo do ambiente μCODE, oferecendo um conjunto mínimo de primitivas voltado à construção de qualquer operação de mobilidade de alto-nível.
- `mucode.util` - Contém recursos para lançamento do servidor μCODE aceitando opções em linha de comando e, mais importante, mecanismos que complementam as capacidades de reflexão da linguagem Java
- `mucode.abstractions` - Contém abstrações de alto-nível construídas sob o μCODE, incluem primitivas para realocar código e estado de várias formas, bem como a

[13] Do inglês: *Remote Dynamic Linking*

implementação de um modelo de agente móvel.

O conceitos de ***group***, ***group handler***, ***μServer*** e ***class space*** constituem o núcleo do μCODE, e são providos pelo pacote `mucode`. Abstrações de alto-nível são construídas com base nesses conceitos, entre elas estão as abstrações oferecidas através do pacote `mucode.abstractions`.

2.6.2 Segurança no Ambiente μCode

Embora seu criador tenha se empenhado em projetar um ambiente que procure não ser inseguro, e reconheça a importância de tais caracteristicas, até o presente momento o μCODE não possui quaisquer mecanismos especialmente projetados com o objetivo de prover segurança (PICCO, 2001b).

Espera-se que a definição simples de uma unidade de migração, o *group*, aliada ao fato de que qualquer outra primitiva de mobilidade pode ser implementada sob ela, facilite ao programador a tarefa de oferecer uma versão segura do μCODE.

3 Segurança em Java

A popularização dos computadores pessoais ocasionou profundas mudanças na sociedade, grande parte das organizações foram obrigadas a rever o modo como conduziam e gerenciavam seus negócios. Assim, no início da década de 90, acreditava-se que os dispositivos eletrônicos seriam a próxima área a sofrer forte influência dos microprocessadores, com o objetivo de criar os chamados equipamentos inteligentes (DEITEL; DEITEL, 2001).

Vislumbrando esse mercado, na época em formação, a empresa americana *Sun Microsystems* investiu em uma pesquisa corporativa interna, que visava à criação de uma linguagem para desenvolvimento de software embutido. A esse projeto foi atribuído o codinome *Green*, sendo que seus trabalhos foram iniciados no ano de 1991. Esse projeto deu origem a uma nova linguagem de programação, baseada na linguagem C/C++, batizada como *Oak*[1] pelo seu criador, James Gosling. Nome esse que foi posteriormente alterado, em função da existência de uma outra linguagem de mesmo nome.

Dessa forma a nova linguagem foi novamente batizada, recebendo dessa vez o nome de **Java**[2] (DEITEL; DEITEL, 2001; NEWMAN, 1997).

A partir de 1993, com o surgimento da WWW - *World Wide Web*, o projeto da linguagem Java ganhou novo enfoque, uma vez que o mercado para dispositivos eletrônicos inteligentes não havia se desenvolvido conforme o esperado pela *Sun*, o que inclusive algumas vezes ameaçou a continuidade do projeto *Green*. Entretanto, os pesquisadores da *Sun*, atentos à popularidade da WWW enxergaram um imediato potencial a ser explorado, a criação de páginas com o chamado conteúdo *dinâmico* (DEITEL; DEITEL, 2001). O anúncio oficial do lançamento da linguagem Java ocorreu em Maio de 1995, a Tabela 8 apresenta as principais características da linguagem Java e que posteriormente serão co-

[1] Carvalho, em inglês
[2] Cidade de origem de um tipo de café importado nos Estados Unidos.

mentadas de modo mais detalhado.

Característica	Comentário
Simples	De forma a ser facilmente programada por grande parte dos desenvolvedores.
Familiar	Para que os atuais desenvolvedores de software tenham condições de aprender a linguagem.
Orientada a Objetos	O que possibilita explorar as vantagens das modernas metodologias de desenvolvimento de software, aplicando-as as aplicações distribuídas cliente/servidor.
Multithreaded	Maior desempenho para aplicações que precisam executar múltiplas atividades de maneira concorrente.
Interpretada	o que permite aproveitar ao máximo as capacidades dinâmica e de portabilidade da linguagem.

Tabela 8: Principais Características - Linguagem Java

Segundo Deitel e Deitel (DEITEL; DEITEL, 2001), nos dias de hoje a linguagem Java tem sido amplamente empregada na criação de páginas para WWW com conteúdo dinâmico e principalmente interativo, no desenvolvimento de aplicativos corporativos de larga escala, no aprimoramento das funcionalidades dos servidores WWW e no fornecimento de aplicativos para dispositivos destinados ao consumidor final, como: telefones celulares, *pagers*, PDA's, entre outros. Segundo Gosling (GOSLING; MCGILTON, 1996), contribuiram também para a popularização da linguagem Java, o fato de que o maciço crescimento da WWW trazer a tona um novo cenário dentro do desenvolvimento de software, demandando novas formas de construção e distribuição de aplicações.

Nos últimos anos, como reflexo do crescente interesse pelos benefícios advindos da exploração da mobilidade de código em aplicações distribuídas voltadas à WWW (seção 2.1), e em função de suas características nativas, a linguagem Java passou a figurar no rol das alternativas ao desenvolvimento desse tipo de aplicação.

Neste contexto, especial atenção tem sido dada às questões relacionadas à segurança dessas aplicações. Em razão disso os mecanismos de segurança implementados e disponibilizados pela linguagem Java têm sido foco de interesse por parte de pesquisadores e desenvolvedores que buscam formas de controlar e garantir a execução dos aplicativos de uma maneira segura.

Nas seções subseqüentes, são comentadas com mais detalhes as principais características da Linguagem Java (Tabela 8) e posteriormente serão descritos os mecanismos de segurança por ela implementados.

3.1 Características Básicas

Java, é uma linguagem orientada a objeto de alto nível que foi projetada para ir ao encontro dos desafios do desenvolvimento de aplicações heterogêneas, voltada a um ambiente distribuído de larga escala, como a WWW. O ambiente de programação Java é composto por um conjunto de ferramentas que inclui um compilador (*javac*), um depurador (*jdb*), entre outras, além de uma máquina virtual que atua como um sistema operacional neutro, a chamada JVM - *Java Virtual Machine* (NEWMAN, 1997; GOSLING; MCGILTON, 1996).

Dessa forma, enquanto em muitas linguagens a execução de um código exige a compilação em código nativo ou em algumas situações a interpretação do código, a linguagem Java diferencia-se justamente pelo fato de exigir a compilação do código-fonte em um código intermediário (*.class*) e a posterior interpretação desse código intermediário.

Durante a compilação é realizada a tradução do código-fonte em uma linguagem intermediária chamada *Java Bytecodes*, que posteriormente é analisada e executada pelo interpretador. Enquanto a compilação do código-fonte ocorre uma vez, a interpretação irá ocorrer a cada vez que o programa for executado (GOSLING; MCGILTON, 1996). A Figura 4 apresenta uma representação do processo de compilação e interpretação de um código Java.

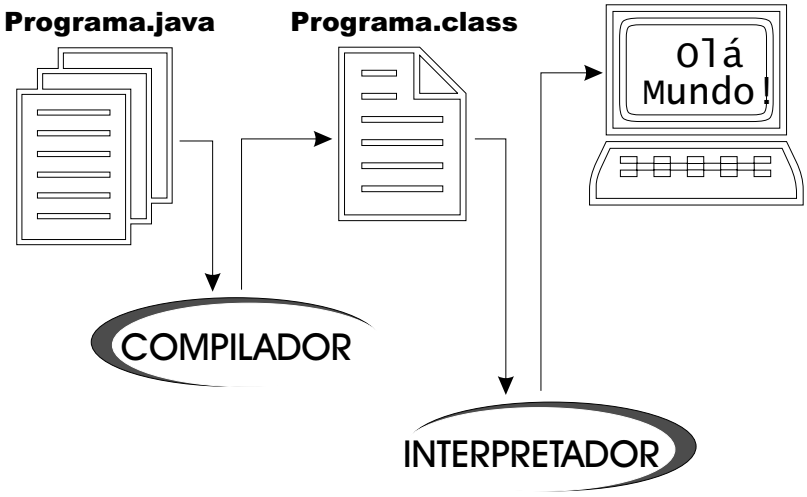


Figura 4: Processo de Compilação - Linguagem Java

Em função das idéias envolvidas com o uso dos *Bytecodes*, o lançamento da linguagem Java foi acompanhado pelo *slogan Write Once, Run Anywhere*[3] que procurava

[3]Tradução: Escreva uma vez, Execute em qualquer lugar.

destacar a portabilidade associada à linguagem. Assim, é possível compilar o código-fonte em qualquer plataforma onde exista um compilador Java, e a partir desse ponto, os *bytecodes* gerados poderão ser executados em qualquer implementação da máquina virtual Java - seja um ambiente Windows, Linux, UNIX-Solaris, Macintosh, entre vários outros, conforme representação da Figura 5.

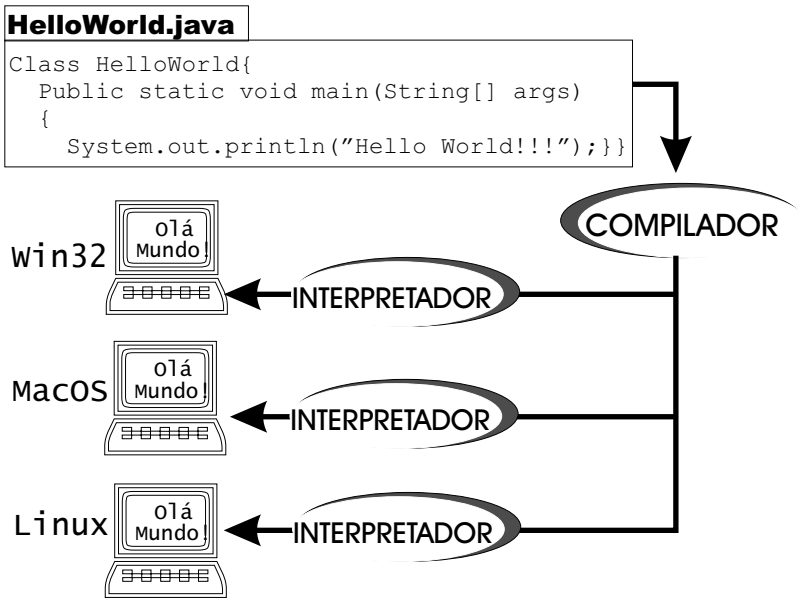


Figura 5: Esquema de Execução - Linguagem Java

De acordo com Newman (NEWMAN, 1997), Java é a primeira linguagem de programação que não esta vinculada a um sistema operacional ou microprocessador específico. Os aplicativos escritos em Java pode ser executados em diversas plataformas, eliminando o problema da incompatibilidade entre sistemas operacionais e suas versões.

A Tabela 9, apresenta os adjetivos-chave que caracterizam a linguagem Java, segundo Newman e Gosling e McGilton (NEWMAN, 1997; GOSLING; MCGILTON, 1996):

Simples	Independente de Arquitetura
Orientada a Objetos	Portável
Distribuído	Alta Performance
Interpretada	Multitarefa
Robusta	Dinâmica
Segura	-

Tabela 9: Linguagem Java - Características

3.1.1 Linguagem Java - Características

A seguir um breve comentário a respeito das principais características da linguagem Java:

Orientada a Objetos e Simples - Uma das principais características da linguagem Java, inclui a simplicidade, uma vez que os principais conceitos da tecnologia Java podem ser facilmente entendidos. Desde sua concepção, as idéias da orientação a objetos permeiam o projeto da linguagem, fazendo com que a tecnologia Java ofereça uma limpa e eficiente plataforma baseada em objetos.(GOSLING; MCGILTON, 1996)

Independente de Arquitetura e Portável - Derivado da natureza distribuída da arquitetura cliente/servidor, um importante recurso da linguagem Java é o suporte a clientes e a servidores em configurações heterogêneas de rede. O que somente é possível porque o compilador Java gera instruções de *bytes* e não binárias, sendo que as instruções de código em *bytes* são fáceis de interpretar em qualquer máquina e são facilmente traduzidas em código de máquina (NEWMAN, 1997; GOSLING; MCGILTON, 1996).O compilador Java foi escrito com a própria linguagem, enquanto seu ambiente de tempo em execução foi escrito em ANSI C (NEWMAN, 1997).

Distribuído - Segundo Newman (NEWMAN, 1997), Java é distribuído devido ao processamento compartilhado da carga de trabalho, uma característica essencial dos aplicativos cliente/servidor. O termo distribuído descreve o relacionamento entre objetos de sistemas, estejam esses objetos em sistemas remotos ou locais.

Alta Performance - Performance é sempre um questão a ser discutida, na linguagem Java essa tarefa é realizada por meio da adoção de um esquema em que o interpretador pode executar com velocidade total sem a necessidade de verificar o ambiente de execução. O coletor de lixo, *garbage collection*, é executado como um *thread* de baixa prioridade, que tem a função de fazer o gerenciamento da memória, por exemplo, liberando o espaço ocupado por elementos que não estão mais em uso. (GOSLING; MCGILTON, 1996).

Interpretada, *Threaded*, Dinâmica - O interpretador Java pode executar os *bytecodes* Java diretamente em qualquer máquina onde exista o interpretador e o sistema *run-time* da linguagem, o que beneficia a agilidade do ciclo de desenvolvimento, normalmente composto de prototipação, experimentação e desenvolvimento rápido (GOSLING;

MCGILTON, 1996). Modernas aplicações baseadas em rede, tipicamente necessitam realizar diversas tarefas de maneira concorrente. A linguagem Java é multitarefa e oferece meios para a construção de aplicações com múltiplas *threads*[4] concorrentes como característica nativa, através do uso de sofisticadas primitivas de sincronização.

Robusta e Segura - A linguagem Java foi projetada para a criação de software altamente confiável, oferecendo uma extensiva verificação em tempo de compilação, seguida por um segundo nível de checagem em tempo de execução. O modelo de gerenciamento de memória é extremamente simples, contudo, elimina vários erros de programação que por vezes prejudicam outras linguagens (GOSLING; MCGILTON, 1996).

A linguagem Java inclui várias bibliotecas, além de métodos para uso dos desenvolvedores na criação de aplicações multi-plataformas, de maneira resumida essas bibliotecas são:

- `Java.lang` - coleção de tipos base que são sempre importados dentro alguma dada unidade de compilação. Sendo o local onde são encontradas as declarações de Objeto e Classe, *threads*, exceções, os tipos de dados primitivos e uma variedade de outras classes fundamentais.

- `Java.io` - arquivos de acesso randômico e *streams*. Encontra-se aqui o equivalente as bibliotecas padrão de entrada/saída presente em vários sistemas. Há também a biblioteca `Java.net`, responsável por prover suporte a *sockets*, *telnet*, interfaces e URL's[5].

- `Java.util` - composto de container e classes utilitárias. Onde é possível encontrar classes como: *Dictionary*, *Hashtable* e *Stack*, entre outras.

- `Java.awt` - o *Abstract Windowing Toolkit* oferece um nível de abstração que permite facilmente portar aplicações a partir de um sistema de gerenciamento de janelas para outro. Esta biblioteca contem classes para componentes básicos da interface tais como: eventos, cores, fontes e controles tais como botões e barras de rolagem.

Vale ressaltar, que os componentes de interface gráfica do pacote `Java.awt` estão diretamente associados com as capacidades de interface da plataforma local. Então, um

[4] Processo de execução da parte de um aplicativo
[5] Do inglês: *Uniform Resource Locator*

programa Java executando em diferentes plataformas pode apresentar uma aparência diferente em cada um dos ambientes utilizados. Por essa razão, atualmente são muito utilizados os chamados componentes ***Swing***, pertencentes ao pacote `javax.swing`, são escritos, manipulados e exibidos completamente em linguagem Java (conhecidos como componentes Java puros) e que por não apresentarem associação com os elementos de interface da plataforma local, permitem que as aplicações apresentem a mesma aparência independemente do sistema operacional utilizado.

por essa razão são também chamados de componentes Java puros.

Por meio dos componentes ***Swing*** o programador pode especificar uma aparência e um comportamento diferente para cada plataforma, ou uma aparência e um comportamento uniforme entre todas as plataformas, sendo possível alterar a aparência e o comportamento enquanto o programa está em execução (DEITEL; DEITEL, 2001).

3.2 Mecanismos de Segurança

Segundo Gritzalis e Iliadis (GRITZALIS; ILIADIS, August 1998), segurança é a prática pela qual individuos e organizações protegem suas propriedades físicas e intelectuais de todas as formas de ataques. A preocupação com questões relacionadas à segurança cresce na mesma proporção em que computadores e as configurações das redes passam a oferecer novos serviços e níveis de acesso, e consequentemente novas oportunidades para interações não-autorizadas e possíveis ocorrências de abuso.

Os riscos relacionados com segurança podem ser subdivididos em quatro categorias básicas: **modificação do sistema**, **invasão de privacidade**, **negação de serviço** e **antagonismo**.(MCGRAW; FELTEN, 1999). Sendo que, as duas primeiras categorias mencionadas são moderadamente bem tratadas pela linguagem Java, enquanto as categorias restantes não o são.

Java foi projetada para criar aplicações altamente confiáveis. Ela oferece extensivo suporte à verificação em tempo de compilação, seguida por um segundo nível de verificação em tempo de execução. Algumas das características da linguagem tendem a guiar os programadores de forma a evitar hábitos de programação que possam trazer problemas, como acontecia com programadores C/C++, com relação em ao gerenciamento de memória, por exemplo. O modelo de gerenciamento de memória da linguagem Java -

onde não há ponteiros - evita uma classe inteira de erros de programação que poderiam deixar as aplicações vulneráveis (GOSLING; MCGILTON, 1996).

Para Gosling (GOSLING; MCGILTON, 1996), o fato de ter sido concebida para operar em ambientes distribuídos, significa que a segurança sempre foi uma das principais preocupações em todo o projeto da linguagem. Com as funcionalidades de segurança projetadas e implementadas na linguagem e o sistema em tempo de execução, o que torna possível construir aplicações não passíveis de invasão.

Em sua forma padrão, Java apresenta um abordagem multicamadas à segurança, sendo que de modo geral, as camadas incluem:(MCGRAW; FELTEN, 1999)

- Acesso restrito ao sistema de arquivos e à rede
- Acesso restrito aos recursos do sistema, tais como: sistema de arquivos, abertura de conexões de rede, entre outros.
- Um conjunto de verificações em tempo de execução e carregamento que permite avaliar se o *bytecode* está em conformidade com as regras, e
- Um sistema para assinatura de código e para atribuição de algum nível de permissões.

As seções subseqüentes apresentam a evolução do modelo de segurança implementado pela linguagem Java, desde seu surgimento na versão 1.0.2 do kit de desenvolvimento da linguagem Java, até o modelo atualmente vigente no Java 2, destacando também a primeira expansão realizada na arquitetura básica de segurança do Java (JDK 1.1).

3.2.1 Arquitetura Básica de Segurança (JDK 1.0.2)

Segundo Gong (GONG, 1999), na versão original da linguagem Java, a arquitetura básica de segurança estava centrada na concepção de permitir a um usuário dinamicamente importar e executar algum código Java sem trazer riscos indevidos ao usuário.

O modelo original de segurança implementava um modelo *sandbox* (item 2.5.1 - p. 24) que impunha um rígido controle sobre a execução de programas Java. Dessa forma, o modelo restringia as ações que certos tipos de programas poderiam ou não executar

(MCGRAW; FELTEN, 1999). Baseado no conceito de aplicações *confiáveis* e *não-confiáveis* apresentado nesse modelo. Em linhas gerais, nas operações não-confiáveis é negado o acesso ao sistema de arquivos e às conexões de redes, sendo as mesmas executadas dentro do chamado *sandbox*. Por sua vez, as aplicações confiáveis, possuiam acesso completo a todos os recursos vitais do sistema, conforme mostra a Figura 6 (GRITZALIS; ILIADIS, August 1998; MCGRAW; FELTEN, 1999).

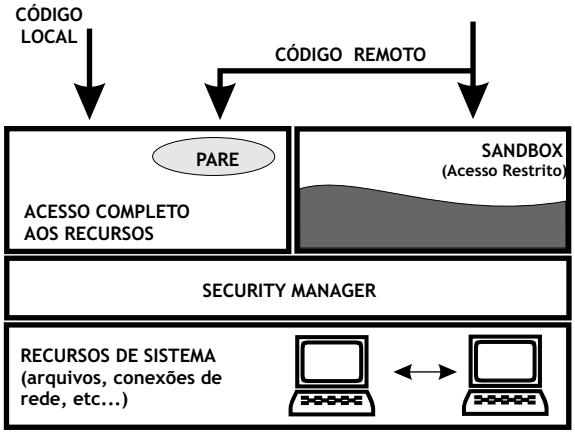


Figura 6: Modelo de Segurança - JDK 1.0.2

O consenso, era de que as *applets* Java, em função de serem transferidas dinamicamente e frequentemente sem a manifestação explícita de concordância do usuário, e também pela dificuldade em se conhecer sua origem e autores, eram códigos ***não-confiáveis*** e portanto deveriam ter sua execução restrita à uma área específica, chamada de *sandbox*. Em razão disso, um *applet* que solicitasse a abertura de uma conexão com a rede, teria essa operação classificada como ***não-permitida*** (GONG, 1999).

Em contra-partida, devido à visão de que são código alocados no mesmo domínio do usuário, as aplicações Java, eram consideradas ***confiáveis***. Sendo assim, se estas aplicações fizessem a mesma solicitação de abertura de uma conexão com a rede, esta seria classificada como uma ***operação permitida*** (GONG, 1999).

O modelo original *sandbox* implementado é composto por três camadas interrelacionadas: o ***Bytecode Verifier***, o ***Class Loader*** e o ***Security Manager***. Para o perfeito funcionamento do sistema de segurança, todas as camadas devem funcionar em conjunto, ou seja, se uma camada falhar, todo o sistema estará comprometido (GRITZALIS; ILIADIS, August 1998; MCGRAW; FELTEN, 1999).

3.2.1.1 *Bytecode Verifier*

A verificação dos arquivos *bytecode* é uma das maneiras pela qual Java analisa automaticamente se um código é confiável ou não-confiável antes de permitir sua execução. Uma vez realizada essa verificação, o código está em condições de ser executado de modo ininterrupto por uma máquina virtual Java (GONG, 1999; MCGRAW; FELTEN, 1999).

O *Bytecode Verifier* não está acessível a programadores nem a usuários, por estar construído junto com a JVM. A ele é atribuída a tarefa de automaticamente examinar o código dos programas assim que estes chegam à máquina virtual. Se alguma inconsistência for detectada, uma exceção será gerada e aquela classe não será executada (MCGRAW; FELTEN, 1999).

Cabe a esse verificador também, garantir a compatibilidade binária entre classes, que podem estar dinamicamente vinculadas apesar de não serem compatíveis. Há regras de compatibilidade que regem a capacidade de mudar o uso de classes e métodos sem quebrar a compatibilidade binária. Por exemplo, é possível adicionar um método a uma classe que é usada por outras classes, contudo, deletar métodos de uma classe que é utilizada por outras classes não é permitido. Regras como essa, são reforçadas pelo *Bytecode Verifier*.

De acordo com McGraw (MCGRAW; FELTEN, 1999) e Gong (GONG, 1999), uma vez que o código tenha passado pelo *Bytecode Verifier*, ele é analisado de modo a garantir que:

- Não seja permitida a simulação do uso de ponteiros, ou seja, nenhum trecho do código pode referenciar diretamente a memória;
- Não haja violações quanto a restrições de acesso;
- Sejam acessados somente objetos pertencentes ao mesmo tipo - por exemplo, garante que objetos `InputStream` serão sempre utilizados como objetos `InputStream` e nunca de alguma outra forma.
- Os métodos são chamados com os devidos argumentos e que não irão ocorrer situações de *overflow* na pilha.
- Não ocorram conversões ilegais, como por exemplo: inteiros para referências a objetos

Assim, cabe ao *Bytecode Verifier*, atuar como um primeiro guardião do modelo de segurança implementado pela linguagem Java. A ele cabe assegurar que cada código transferido seja executado segundo regras de segurança.

3.2.1.2 *Class Loader*

Um dos principais objetivos da linguagem Java é fazer código verdadeiramente móvel, ou seja, possibilitar que dinamicamente um código oriundo de algum ponto externo ao domínio do usuário seja carregado e executado. Na linguagem Java, o código é carregado por meio de um *Class Loader*, independetemente se o código esteja armazenado localmente ou em algum ponto da rede.

Segundo McGraw e Felten (MCGRAW; FELTEN, 1998), duas funções são executadas pelo *Class Loader*, a primeira é quando a máquina virtual necessita carregar o *bytecode* para uma classe em particular, neste caso ele possui a responsabilidade de encontrar o *bytecode*. Isso pode ser feito de diversas maneiras, cada *Class Loader* pode usar seus próprios métodos na busca do *bytecode* requerido: ele pode carregá-los de um disco local, requerê-los de algum ponto da rede através de algum protocolo, ou apenas criar um *bytecode* on the fly.

A segunda função, diz respeito à definição dos chamados *Namespaces* visto pelas diferentes classes, e como estes se relacionam entre si. Os *Namespaces* permitem que classes Java tenham diferentes visões do contexto, dependendo de onde elas se originam (MCGRAW; FELTEN, 1998).

Vale ressaltar, que ao receber um *bytecode* e antes de executá-lo o *Class Loader* irá invocar o *Verifier* - que será o responsável pela autorização ou não do código importado. O *Class Loader* constitui uma peça chave dentro do modelo de segurança implementado pela linguagem Java. (GRITZALIS; ILIADIS, August 1998; MCGRAW; FELTEN, 1999)

3.2.1.3 *Security Manager*

É a terceira camada do modelo básico de segurança da linguagem Java, e tem a tarefa de restringir como uma classe utiliza as chamadas à API Java. O *Security Manager* é o responsável pela implementação de uma boa porção do modelo de segurança (MCGRAW; FELTEN, 1999).

Cabe a esta camada rastrear quem está habilitado a realizar operações consideradas *perigosas*. Em geral, a maioria das operações não estão habilitadas quando requisitadas por códigos considerados não-confiáveis. Em contra partida, os códigos confiáveis possuem total liberdade de execução.

O *Security Manager* é um objeto único em Java que executa verificações em tempo de execução em métodos considerados *perigosos*, sendo requisitado toda vez que um desses métodos tenta ser executado.

Para McGraw (MCGRAW; FELTEN, 1999), o uso do *Security Manager* ocorre da seguinte maneira:

- Um programa em Java, realiza uma chamada à API Java para a realização de uma operação potencialmente perigosa;
- A API Java, requisita então permissão ao *Security Manager* para execução da operação;
- O *Security Manager* atende a requisição com uma `SecurityException`, caso a execução da operação seja negada. Esta exceção é propagada a todo o programa.
- Se a operação é *permitida* (item 3.2.1 - p. 40), o *Security Manager* não retorna nenhuma exceção, e a API Java então executa a operação requisitada e retorna normalmente.

A seguir McGraw e Felten (MCGRAW; FELTEN, 1999) destacam algumas das tarefas que se encontram sob a responsabilidade do *Security Manager*:

- Prevenir instalação de novos *Class Loaders*, cujo trabalho é manter os *Namespaces* adequadamente organizados.
- Proteger *threads* e grupos de *threads*, evitando ataque das chamadas *threads* maliciosas.
- Controlar a execução de outros programas.
- Controlar a habilidade de desligar a máquina virtual, JVM.
- Controlar o acesso a outros processos.

- Controlar o acesso aos recursos do sistema, tais como: filas de impressão, propriedades de sistema, janelas, entre outros.
- Controlar as operações do sistema de arquivo.
- Controlar as operações de rede (`socket`), tais como: `connect` e `accept`.
- Controlar o acesso aos pacotes Java.

Cada máquina virtual, JVM, possui somente um *Security Manager* instalado a cada vez, sendo que uma vez instalado ele não poderá mais ser desinstalado, exceto por meio da re-inicialização dessa JVM.

As mudanças realizadas no modelo de segurança da linguagem Java, quando do lançamento das versões **JDK 1.1** e do **Java 2**, afetaram radicalmente o papel desempenhado pelo *Security Manager* (MCGRAW; FELTEN, 1999).

3.2.2 Extensão ao Modelo Original - *Signed Code* (JDK 1.1)

O JDK 1.1, lançado na primavera de 1997, apresentou o conceito de código assinado, ou *signed code*, que gerou profundas mudanças no modelo originalmente implementado na linguagem Java.

O modelo *sandbox* originalmente implementado, considerava que qualquer código remoto, entenda-se *applet* Java, era automaticamente *não-confiável* e consequentemente sua execução estava confinada dentro de um *sandbox*. Tal restrição, apesar de contribuir para um ambiente seguro, causava muitos transtornos aos desenvolvedores de aplicações (GONG, 1999).

Assim, visando a dar mais flexibilidade, o JDK 1.1 passou a incluir suporte ao recurso de **assinaturas digitais**. Permitindo que após o desenvolvimento, uma classe Java pudesse ser assinada e armazenada junto com suas assinaturas em um arquivo JAR - *Java Archive*, por exemplo. Assim, após a transferência do código as assinaturas dessa classe poderiam ser verificadas e reconhecidas como *confiáveis*, o que possibilitou que *applet´*s Java tivessem o mesmo tratamento das aplicações *confiáveis*, obtendo acesso completo aos recursos do sistema, conforme mostra a Figura 7 (GRITZALIS; ILIADIS, August 1998; MCGRAW; FELTEN, 1999; GONG, 1999).

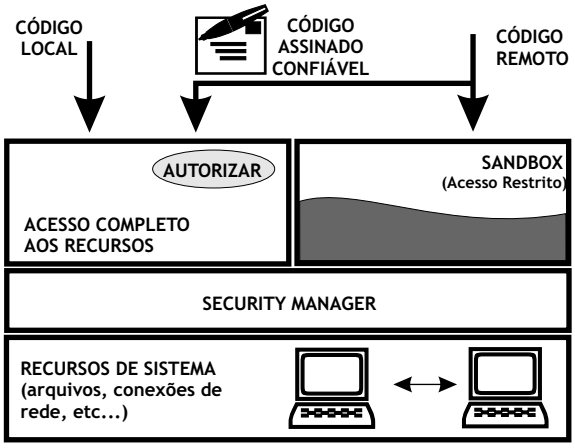


Figura 7: Modelo de Segurança - JDK 1.1

Segundo McGraw e Felten (MCGRAW; FELTEN, 1999), a introdução do *code signing* no JDK 1.1 implicou em uma mudança de *postura* por parte do modelo *sandbox* do Java, que originalmente aplicava igualmente um conjunto de regras a todos os *applets* Java. Agora, encontra-se um sistema mais flexível e que pode ser expandido e personalizado *applet* a *applet*.

3.2.2.1 Aprimoramentos na Segurança - JDK 1.1

Apesar do considerável aumento na complexidade do modelo de segurança, o principal objetivo da introdução de tais modificações foi obter um melhor controle sobre a segurança dos códigos móveis. E para tanto, com a introdução do *code signing* e a expansão do modelo de confiança, espera-se alcançar esse objetivo por meio das seguintes habilidades (MCGRAW; FELTEN, 1999):

- Garantir privilégios quando eles são necessários;
- Ter código operando somente com os privilégios necessários;
- Realizar um gerenciamento detalhado das configurações do sistema de segurança.

Dessa forma, a política de segurança introduzida é instanciada no momento da inicialização da máquina virtual Java e pode ser alterada *a posteriori* através do uso de métodos seguros. Permissões não são mais garantidas às classes mas sim a domínios, ou seja, a todos os objetos relacionados ao objeto autorizado pelo sistema (MCGRAW; FELTEN, 1999; GRITZALIS; ILIADIS, August 1998).

Do ponto de vista de segurança, a mais importante mudança introduzida no JDK 1.1 foi a adição da autenticação e mecanismos simples de controle de acesso, que contam inclusive com o uso de criptografia. A seguir listamos os principais aprimoramentos realizados:

- **Criação da API Cripto** - inclui tanto uma API como alguns pacotes para implementação de algumas funcionalidades por trás da API. As classes do pacote `java.security` tem dois propósitos: (1) prover métodos criptográficos que possam ser usados para implementação do modelo de segurança e (2) dar aos desenvolvedores o suporte as funcionalidades de criptografia, para criação de aplicações seguras.
- **Certificado** - Outro recurso introduzido foi o certificado tecnológico baseada no padrão aberto X.509v3. Certificação oferece um mecanismo de autenticação pelo qual um nó da rede pode seguramente reconhecer outro, conseqüentemente, pontos da rede que tem a capacidade de se reconhecerem têm a oportunidade de estabelecer um relacionamento de confiança. Quando uma conexão SSL - *Secure Socket Layer* é inicializada entre duas máquinas, elas *se reconhecem* através da troca de certificados.
- **Comunicação Segura** - Java 2 passou a incluir um pacote para comunicação através do protocolo SSL - *Secure Socket Layer*. Dessa forma, é capaz de prover um canal seguro de comunicação pelo uso de encriptação.

Segundo Gong (GONG, 1999), uma das principais contribuições dessa nova proposta, foi a inclusão da idéia de código assinado, ou *code signing*, o que possibilitou flexibilizar o desenvolvimento e execução das aplicações Java, independente de qual o tipo da aplicação: *applets* ou *aplicações* Java. Isso acabou por alterar também o conceito de código *confiável* e *não-confiável*.

Tanto o modelo *sandbox* original como o modelo baseado em confiança introduzido posteriormente, foram novamente alterados quando do lançamento da nova arquitetura de segurança no **JDK 1.2**, mais conhecido pela denominação **Java 2**.

3.2.3 Um Novo Modelo de Segurança (Java 2)

O lançamento do chamado Java 2, versão 1.2 do JDK, ocorrido em fevereiro de 1997 apresentou uma nova arquitetura de segurança para a linguagem Java. As melhorias

adicionais a essa versão atenderam a demanda por políticas de controle de acesso flexíveis e granulares, que fosse extensíveis e escaláveis (GONG, 1999; MCGRAW; FELTEN, 1999).

Dentro as razões que motivaram a definição de uma nova arquitetura de segurança para a linguagem Java, destaca-se (MCGRAW; FELTEN, 1999):

- As restrições aplicadas eram fatores muito limitantes;
- Separação insuficiente entre política de segurança e os mecanismos para aplicação;
- Dificuldade de extensão das verificações de segurança;
- Códigos instalados localmente não devem ser considerados confiáveis sem uma análise mais criteriosa;
- Fragilidades no sistema interno de segurança.

Dessa forma, assume-se que o modelo de segurança implementado pela linguagem Java, evoluiu do chamdado *modelo preto e branco* baseado em confiança para um *modelo de escala de cinza*. Visto que não mais estão presentes a idéia de códigos confiáveis e não-confiáveis baseados somente na assinatura digital.

A nova arquitetura de segurança, se vale de uma política de segurança para decidir que permissões individuais de acesso estão garantidas para a execução de um código. Estas permissões estão baseadas nas características apresentada pelo código: quem o assina, qual a sua origem, se ele está assinado digitalmente. Posteriormente, tentativas de acesso a recursos protegidos irão invocar verificações de segurança que têm a função de comparar as permissões necessárias com as permissões garantidas a quem está solicitando o recurso. Caso haja garantia das permissões o acesso será permitido, caso o conjunto de permissões atribuídas seja insuficiente o acesso será negado (GONG, 1999).

Gong (GONG, 1999) destaca que no Java 2, se não houver a definição explícita de uma política de segurança, então o procedimento padrão é aplicar o modelo *sandbox* clássico implementado no JDK 1.0 e JDK 1.1. A Figura 8 apresenta a evolução da arquitetura de segurança na linguagem Java (MCGRAW; FELTEN, 1999).

Essa nova arquitetura resultou na inclusão de novos mecanismos de proteção: políticas de segurança, permissões de acesso, proteção de domínios, verificação de controle

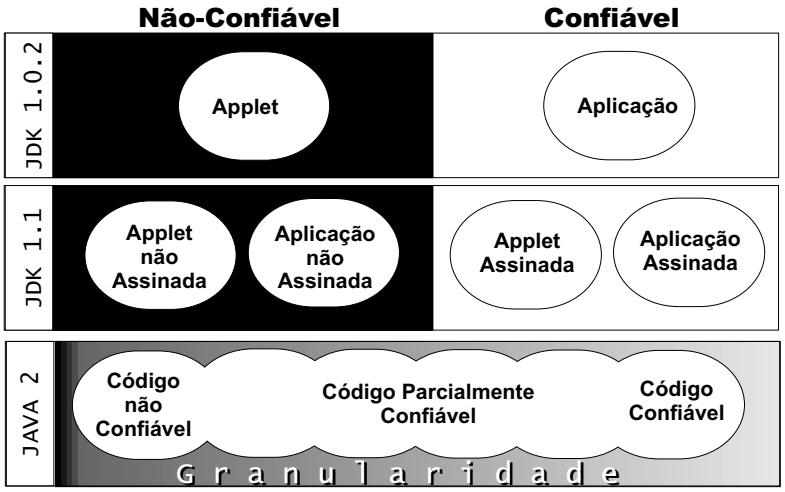


Figura 8: Evolução da Arquitetura de Segurança - Linguagem Java

de acesso, operações privilegiadas e carregamento e resolução de classes Java (GRITZALIS; ILIADIS, August 1998; MCGRAW; FELTEN, 1999).

Essencialmente, o modelo em três camadas ainda permanece sendo a base do modelo de segurança implementado pela linguagem, contudo, tornou-se um modelo híbrido e extendido incluíndo a noção de *granularidade*, ou seja, graus de confiança e administração de permissões baseadas em assinaturas digitais.

Por essa razão, a partir dessas modificações, a decisão de confiar ou não em um código poderá ser influenciada pela informação de quem o certificou, o *voucher*[6]. Assinaturas digitais são usadas nesse cenário justamente para determinar quem certifica um determinado código (MCGRAW; FELTEN, 1999).

Além disso, Java 2 aumenta a complexidade desse cenário, ao adicionar o chamado **controle de acesso**, que combinado com o *code signing* permitirá a saída gradual das aplicações, até então, não-confiáveis do *sandbox*. O que em parte, torna vago o significado originalmente definido para o *sandbox*, visto que flexibiliza o conceito *branco* e *preto* até então adotado,

Assim sendo, torna-se possível que um código, não-confiável até aquele momento, consiga executar operações às quais antes lhe seriam negado o acesso. Caso, por exemplo, de um *applet* Java que foi projetado para atuar em uma Intranet. A ele pode ser garantida a permissão de escrever e ler em um banco de dados de uma empresa enquanto ele for assinado pelo administrador do sistema.

[6] Testemunha, fiador, certidão.

A Figura 9 apresenta uma representação do arquitetura de segurança implementada no Java 2:

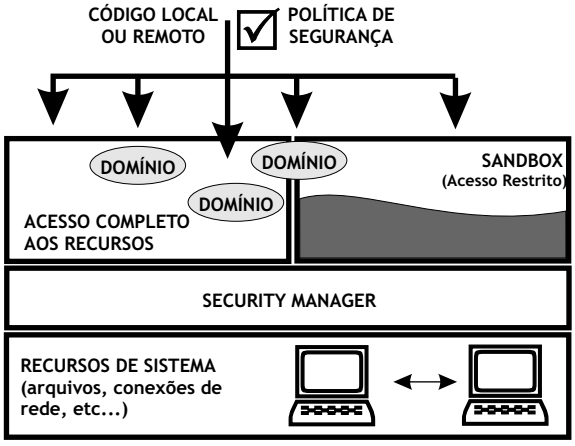


Figura 9: Modelo de Segurança - JDK 1.2 / Java 2

Segundo McGraw (MCGRAW; FELTEN, 1999), a nova arquitetura de segurança apresenta as seguintes habilidades:

- **Graus de Controle de Acesso** - A habilidade de especificar que código com permissões adequadas está em condições de sair gradualmente das condições do *sandbox*, por exemplo, um *applet* assinado por uma chave de confiança pode estar habilitado a criar conexões de rede arbitrariamente.
- **Política de Segurança Configurável** - A habilidade dos construtores de aplicação e usuários Java em configurar e gerenciar políticas complexas de segurança.
- **Estrutura de Controle de Acesso Extensível** - A habilidade de classificação de permissões e agrupar cada permissão de acordo com a política
- **Verificação de Segurança para todos programas em Java** - A partir do conceito de que o código construído deve ser completamente confiável.

Vale ressaltar que as três primeiras dessas habilidades não são verdadeiramente novas para a linguagem Java. Por ser uma poderosa linguagem de programação, é sempre possível implementar políticas de segurança complexas, configuráveis e extensíveis baseadas no controle de acesso. O que foi feito pelo Java 2 foi facilitar essa tarefa para aqueles que não são usuários avançados da linguagem (MCGRAW; FELTEN, 1999).

3.2.3.1 Novos Mecanismos de Segurança - Java 2

Os novos mecanismos de segurança implementados pela linguagem Java, versão 1.2, são apresentados a seguir (MCGRAW; FELTEN, 1999):

3.2.3.2 Identificação

Cada trecho de código necessita de uma identidade específica que serve como base para decisões de segurança, assim, no Java 2, cada uma dessas partes tem duas definições de identidade: origem e assinatura. Estas duas características são representadas na classe `java.security.CodeSource` que permite o uso de caracteres curingas para denotar "qualquer lugar" para a origem e "não assinado" para a assinatura (MCGRAW; FELTEN, 1999).

3.2.3.3 Permissões

Solicitação para executar uma operação em particular, provavelmente considerada perigosa, pode ser encapsulada como uma permissão. A classe abstrata `java.security.Permission` classifica e parametriza um conjunto de permissões de acesso garantindas para classes (MCGRAW; FELTEN, 1999).

As boas práticas de programação ditam que uma permissão de classe deve pertencer ao pacote em que ela é usada e não à uma classe específica. Java 2 define métodos de acesso e parâmetros para muitos dos recursos controlados pela máquina virtual, as permissões incluem:

- `java.io.FilePermission` - para o acesso ao sistema de arquivos;
- `java.io.SocketPermission` - para o acesso a rede;
- `java.io.PropertyPermission` - para as propriedades Java;
- `java.io.RuntimePermission` - para o acesso ao recursos em tempo de execução;
- `java.io.NetPermission` - para autenticação;
- `java.awt.AWTPermission` - para o acesso aos recursos gráficos como as janelas;

Aplicações Java plenamente confiáveis possuem condições de adicionar novas categorias de permissões.

3.2.3.4 *Implies*

Cada classe `Permission` deve incluir o método abstrato *implies*. A idéia é clara: ter a permissão `x` automaticamente implica em ter a permissão `y`. Uma pemissão `x` implica em outra permissão `y` se e somente se tanto o destino de `x` implica no destino de `y` e se a ação de `x` implica na ação de `y` (MCGRAW; FELTEN, 1999).

3.2.3.5 Política de Segurança

Políticas de segurança em Java 2, podem ser ajustadas tanto pelo usuário como pelo administrador do sistema, e são instanciadas a partir da classe `java.security.Policy`.

As políticas de segurança são um mapeamento realizado a partir da identidade, de forma a ajustar as permissões de acesso garantidas ao código.

Exemplo de um objeto `policy` é mostrado aqui na forma de texto:

```
grant CodeBase "https://www.rstcorp.com/users/gem", SignedBy "*" {
 permission java.io.FilePermission "read, write", "/applet/tmp/*";
 permission java.net.SocketPermission "connect", "*.rstcorp.com";
};
```

Nesta política foi determinado que qualquer aplicação proveniente do sítio `"www.rstcorp.com/users/gem"`, seja ela assinada ou não-assinada, poderá ler e escrever qualquer arquivo no diretório `/applet/tmp/*` bem como fazer conexões do tipo *socket* com qualquer máquina pertencente ao domínio `rstcorp.com` (MCGRAW; FELTEN, 1999).

3.2.3.6 Domínios de Proteção

Segundo a *Sun Microsystems*, classes e objetos no Java 2 pertecem a domínios de proteção. O que de fato é apenas um nome fantasia para denotar um grupo de classes

que devem ser tratadas de mesma forma por terem a mesma origem e serem assinadas pela mesma entidade.

Permissões são garantidas a domínios de proteção e não diretamente a classes ou objetos, a classe `java.security.ProtectionDomain` é privada em seu pacote sendo usada internamente para implementar proteção de domínio. Políticas de segurança do sistema especifíca que domínios de proteção deverão ser criados e quais permissões devem ser garantidas (MCGRAW; FELTEN, 1999).

3.2.3.7 Controle de Acesso

A classe `java.security.AccessController` implementa um algoritmo de inspeção de pilha, qualquer código é permitido solicitar essa classe, que executa uma inspeção dinâmica na pilha relevante de *threads* em execução. O método usado para implementar a verificação é `checkPermission`, que se utiliza como seu argumento um objeto `Permission`. Se não ocorrer nenhum retorno, a permissão é garantida e a potencialmente perigosa computação pode prosseguir. Se a chamada falhar, um `AccessControlException` é gerado (MCGRAW; FELTEN, 1999).

3.2.3.8 Privilégios

É através do uso das primitivas `beginPrivileged` e `endPrivileged` que uma parte privilegiada do sistema, isto é, uma parte que tem permissão para realizar tarefas como por exemplo acesso a arquivos. Se utiliza para garantir temporariamente permissão para um código menos confiável.

A idéia é encapsular operações potencialmente perigosas que exigem privilégios *extra* em pequenos e contidos blocos de código. As bibliotecas Java fazem internamente uso constante dessas chamadas, mas código de aplicações parcialmente confiáveis escritos para o modelo implementado pelo Java 2 irão requerer seu uso também.

O uso correto das primitivas do JDK exige o uso do bloco padrão `try/finally` como se segue (MCGRAW; FELTEN, 1999):

```
(código normal) try{
    AccessController.beginPrivileged();
```

```
    (coloque o código perigoso aqui)
}finally {
    AccessController.endPrivileged();
} (mais código normal)
```

Segundo McGraw (MCGRAW; FELTEN, 1999), o Java 2 claramente introduz mudanças significativas no cenário da segurança implementada na linguagem Java. Com as principais mudanças da arquitetura de segurança da linguagem, vieram também novas importantes responsabilidades, sendo a mais importante a política para criação e gerenciamento de código móvel.

Durante muitos anos, os desenvolvedores Java têm procurado alguma maneira em que menos restrições necessitem ser colocadas em seus *applets*. Ao mesmo tempo que, gerentes e muitas empresas tem procurado por maneiras de gerenciar código - não apenas código móvel, mas qualquer código) de forma mais segura. E a linguagem Java oferece uma resposta poderosa a essas necessidades.

3.3 Impactos no Modelo Original

Para McGraw (MCGRAW; FELTEN, 1999), as principais alterações sofridas pelo modelo de segurança originalmente implementado pela linguagem Java em relação ao atual modelo disponibilizado pela versão 1.2 são:

- Revisão do *Security Manager*.
- *Class Loader* Seguro.
- *Sandbox*
- Permissões

3.3.1 Revisão do *Security Manager*

Até a versão 1.1 do JDK o *Security Manager* invocava diretamente o método `check` para controlar o acesso a recursos considerados perigosos. Este método era responsável por avaliar as solicitações, garantindo ou negando acesso.

O **Novo *Security Manager*** no Java 2 ainda oferece suporte ao uso do método `check`, mas agora muitas das chamadas são realmente implementadas para fazer uso dos objetos `AccessController` e `Permission`, sempre que possível (MCGRAW; FELTEN, 1999).

Vale ressaltar, que o *Security Manager* não foi totalmente dispensado por questões de compatibilidade e por se tratar de uma economicamente inviável para a linguagem Java, ou seja, privilegiou-se a compatibilidade com as aplicações já existentes.

3.3.2 *Class Loader* Seguro

Java 2 introduziu a classe `java.security.SecureClassLoader`, que é a concreta implementação da classe abstrata `Class Loader`. Ele rastreia o código fonte e assinaturas de cada classe, e então determina classes para o domínio de proteção.

Todo código Java é carregado por um *Class Loader* seguro, exceto para códigos carregados pelo *Primordial Class Loader* seja de forma direta ou indireta.(MCGRAW; FELTEN, 1999)

Segundo Gong (GONG, 1999), o *Primordial Class Loader* é um tipo de *Class Loader* geralmente escrito em uma linguagem nativa, como a linguagem *C*, e que não se manifesta diretamente dentro do contexto do Java. O *Primordial Class Loader* frequentemente faz o carregamento de classes a partir do sistema de arquivos, de um forma dependente da plataforma utilizada.

3.3.3 *Sandbox*

Com a evolução do modelo de segurança, o *sandbox* tornou-se bem mais complexo e deixou de ser baseado apenas na distinção entre código do tipo remoto (*applets*) ou local (aplicações), como no modelo original, é possivel (e desejável) forçar que as aplicações Java também sejam executadas pelo mecanismo *sandbox*, o que implica que o código das aplicações podem ser feitos de maneira coerente com as políticas de segurança localmente definidas (MCGRAW; FELTEN, 1999).

Java 2 oferece um mecanismo para realizar essa tarefa através do uso da classe `java.security.Main`. A implementação irá assegurar que aplicações locais armazenadas

no `java.app.class.path` sejam carregadas em um *Secure Class Loader*.

3.3.4 *Permissões*

Com a introdução do novo modelo, também foi possível adicionar novas permissões ao Java, que são moldadas de acordo com as necessidades especificas. As novas classes de permissão que foram criadas deverão ser armazenadas no pacote da aplicação onde eles se aplicam (MCGRAW; FELTEN, 1999).

McGraw (MCGRAW; FELTEN, 1999), conclui que mundanças, como a introdução do conceito de permissões, trazidas pelo **Java 2** provocaram mudanças significativas ao escopo de segurança dentro da linguagem Java - ficando muito distante os tempos em que a política de segurança baseava-se apenas no modelo "*branco*" e "*preto*". Acompanhando a evolução da arquitetura, novas e importantes responsabilidades foram criadas, dentre as quais destacamos a criação e gerenciamento de políticas para tratamento de código móvel.

4 Ambiente μCode Seguro

Conforme descrito no Capítulo 2 (seção 2 - p. 5), o atual esquema de funcionamento do ambiente μCODE não oferece mecanismos que permitam o controle efetivo das ações ou recursos utilizados, por uma aplicação. Essa ausência se justifica em função do escopo estabelecido à época da criação do ambiente, que procurou privilegiar os mecanismos de suporte à mobilidade de código, contudo, seu projeto buscou facilitar uma posterior implementação de tais mecanismos (PICCO, 2001b).

Em um primeiro momento, a concepção do mecanismo de segurança para o ambiente μCODE passa pelo entendimento das diferentes necessidades de programadores, administradores e usuários finais, uma vez que eles desempenham diferentes papéis no contexto de uma aplicação baseada no uso de Sistemas de Código Móvel. Observa-se que muitas das discussões são fomentadas em torno das preocupações com a execução de códigos cuja procedência, muitas vezes, está além dos domínios do usuário.

Vale ressaltar que o reconhecimento de um código como ***confiável*** ou ***não-confiável***, é apenas uma parte do processo de segurança de uma aplicação, não sendo suficiente como garantia de segurança, uma vez que um código tido como confiável pode vir a realizar operações danosas, ou então consideradas proibidas. Dessa forma, em um segundo momento, independentemente do grau de confiança atribuído a uma aplicação, a partir do momento em que a aplicação esteja migrando pelas diversas máquinas de uma rede de computadores, também se faz necessário a definição de políticas de segurança que permitam disciplinar o acesso a recursos e informações, ou seja, que tornem possível estabelecer associações entre os usuários de uma aplicação e as ações, ou operações, a eles permitidas.

Neste cenário, uma vez compreendidas as implicações a respeito da segurança envolvida na utilização do ambiente μCODE, ficou estabelecido que o mecanismo de segurança ora implementado estaria baseado nos conceitos de Assinatura de Código e Per-

missões, ambos disponibilizados através da Linguagem Java. Esse mecanismo possibilita que os usuários, em especial os administradores e programadores, tenham condições de realizar um efetivo controle sobre a forma como as aplicações baseadas neste ambiente são executadas, viabiliando o estabelecimento de formas para verificação da confiabilidade de um código e posteriormente o controle das ações realizadas e os recursos por ele utilizados.

A implementação de tal mecanismo deve ter pouco impacto sobre os usuários finais do ambiente μCode, sendo a eles exigido apenas que informem, juntamente com os parâmetros requeridos por uma aplicação, a sua identificação e chave privada. Além de determinar as relações de confiança, caberá também aos administradores definir, por meio da atribuição de permissões, as políticas de segurança que serão aplicadas aos usuários durante a execução da aplicação. Por fim, aos programadores, é dada maior flexibilidade na construção e implementação das aplicações, visto que agora há a possibilidade de uso de versões seguras tanto do ambiente como das abstrações oferecidas pelo μCode.

De modo resumido, a versão segura do ambiente μCode está baseada na construção e utilização de uma unidade de migração digitalmente assinada, ou seja, uma versão assinada do *Group*, e na atribuição de permissões que possibilitem controlar a execução das aplicações no ambiente. Assim, baseado nos conceitos que formam a arquitetura do ambiente μCode (seção 2.6.1 - p. 28) pode-se subdividir o mecanismo de segurança em três partes:

- **Chave Pública e Chave Privada** - A assinatura digital de um *Group* será realizada com base nos conceitos de chave pública e privada, dessa forma é necessário a geração de um par de chaves com base em algum algoritmo de encriptação.
- ***SignedGroup*** - Versão adaptada da unidade de migração original do ambiente μCode, onde, além do armazenamento de classes e objetos, serão agora incluídas informações sobre a assinatura do grupo e a identidade do emissor, o que posteriormente permitirá seu reconhecimento e a validação de sua origem nos demais servidores, ***μServer***, que compõem o domínio de aplicação.
- **Permissões** - Possibilita a associação entre uma unidade de migração digitalmente assinada, o *SignedGroup*, e o conjunto de ações, e recursos que um determinado usuário possui permissão para realizar. Assim, além de complementar o mecanismo de segurança, disciplina o acesso às informações e aos recursos.

As seções subseqüentes descrevem em detalhes a implementação da versão segura do ambiente μCODE.

4.1 Chave Pública e Chave Privada

A assinatura digital de um código exige que, além dos dados a serem assinados, também participem do processo de assinatura duas outras informações: a chave Pública e a chave Privada. Uma assinatura digital é composta por uma cadeia de bits calculada a partir da informação a ser assinada, em conjunto com o par de chaves pertencente a uma determinada entidade. As Figuras 10 e 11 apresentam um esquema do processo de assinatura digital de um código, tanto do ponto de vista do emissor como do receptor.

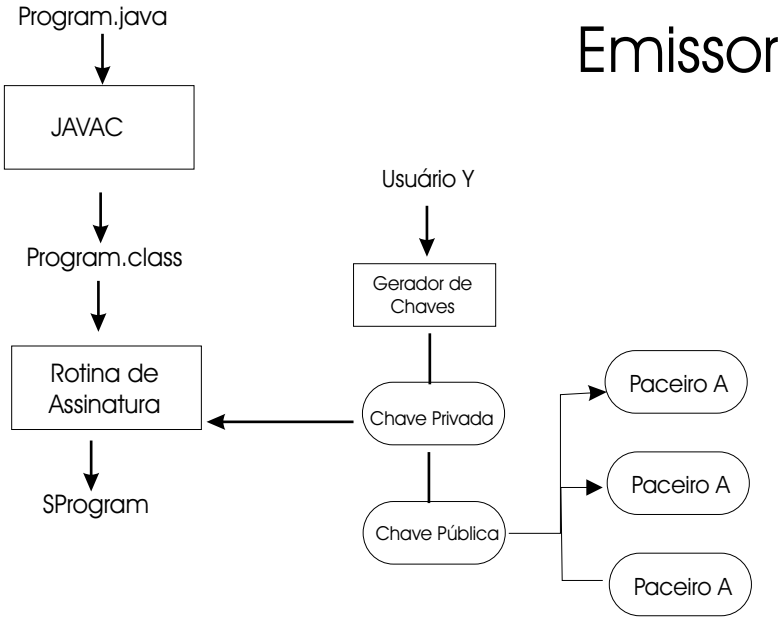


Figura 10: Processo de Assinatura Digital- Emissor

Dessa forma, de acordo com a Figura 10 o usuário que desejar enviar um código digitalmente assinado a uma outra entidade, com a qual mantenha algum tipo de vínculo, precisa, além de criar o programa, ter em mãos um par de chaves válido - o que pode ser obtido por meio de diversas ferramentas. Assim, após a compilação do programa, os *bytecodes* pode ser submetidos a uma rotina para assinatura digital, o que na prática significa utilizar uma das aplicações disponíveis no próprio kit de desenvolvimento da linguagem Java ou então através da programação de rotinas. Além do programa compilado, participa também do processo de assinatura a chave privada do usuário em questão.

Vale ressaltar, que uma vez gerado o par de chaves, este usuário deve distribuir a chave pública às diversas entidades com quais mantem algum tipo de relacionamento. Do contrário, torna-se impraticavel a execução da aplicação no local de destino.

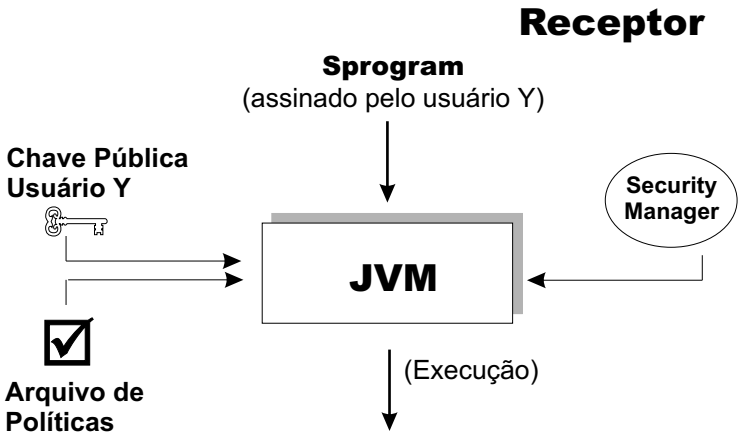


Figura 11: Processo de Assinatura Digital - Receptor

De acordo com a Figura 11, a execução de um código digitalmente assinado, exige que o receptor de tal código tenha previamente em mãos a chave pública do emissor. Essa informação será posteriormente validada e confrontada com a assinatura recebida em conjunto com o código. O controle das ações realizadas por esse código será realizado através dos mecanismos de segurança da linguagem Java, o *Security Manager* e o Arquivo de Políticas, cabendo ao primeiro controlar a execução da aplicação de acordo com as regras especificadas pelo segundo.

Entende-se por chave pública, um número associado a uma entidade em particular e que deve ser de conhecimento de todos aqueles que desejam estabelecer interações confiáveis com tal entidade. Por sua vez, a chave privada, também é um número, porém de conhecimento restrito a uma entidade em particular, que a utilizará durante o processo de assinatura digital.

Dessa forma, no contexto de uma versão segura para o ambiente μCODE, a utilização do recurso de assinatura digital possibilita verificar e validar a origem de um *Group* durante a transferência entre servidores *μServer*. Para tanto, é necessário que uma vez estabelecida uma relação de confiança, as chaves públicas de todos aqueles usuários reconhecidos como confiáveis estejam presentes nos servidores da aplicação. Da mesma forma, esses usuários farão uso de sua chave privada para possibilitar a criação de uma versão assinada da unidade de migração do ambiente μCODE, o *Signed Group*, o que será comentado em detalhe nas seções subseqüentes.

A Figura 12 apresenta a forma como as chaves serão distribuídas na versão segura do ambiente μCODE.

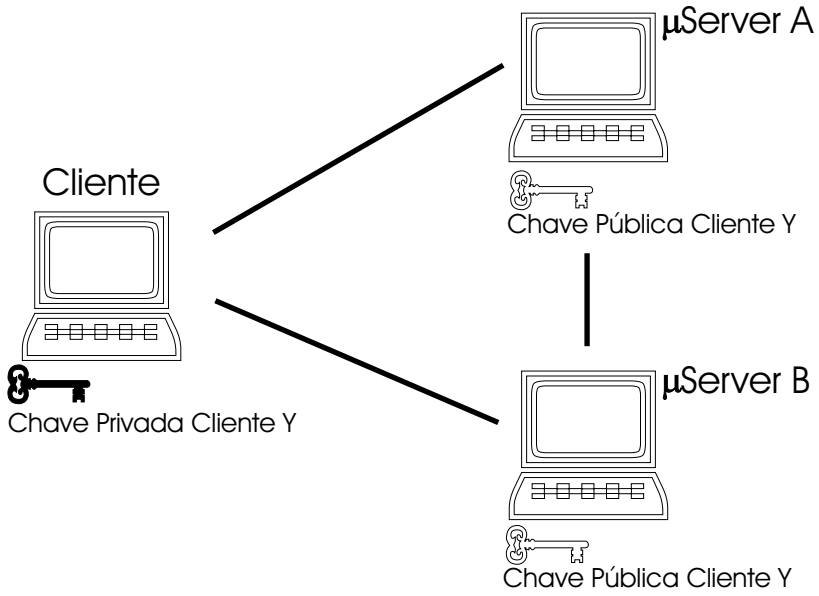


Figura 12: Esquema de Distribuição da Chaves - versão segura.

Em geral, a chave privada fica em poder do usuário sendo mantida em sigilo, porém está sempre associada a uma única chave pública, que ficará de posse daqueles com quem esse usuário mantém alguma forma de interação. Além da possibilidade de implementação de programas para a geração das chaves, os usuários encontram a sua disposição outras ferramentas que também oferecem suporte à geração da chave pública e da chave privada, como é o caso do aplicativo `Keytool`, distribuído juntamente com o, J2SDK, kit de desenvolvimento da linguagem Java.

4.2 *Signed Group*

Em sua essência, o funcionamento do ambiente μCODE está baseado em três conceitos anteriormente mencionados: *Group*, *Group Handler* e *Class Space*. Um *Group* desempenha o papel da unidade de migração no ambiente, permitindo que os programadores tenham a mão um container passível de ser arbitrariamente preenchido com classes e objetos, incluindo objetos do tipo threads, e que posteriormente podem ser remetidos a servidores *μServer* (Figura 13).

Por sua vez, tais servidores atuam em conjunto com o *Group Handler*. Dessa forma, enquanto cabe ao servidor *μServer* oferecer suporte aos mecanismos necessários à

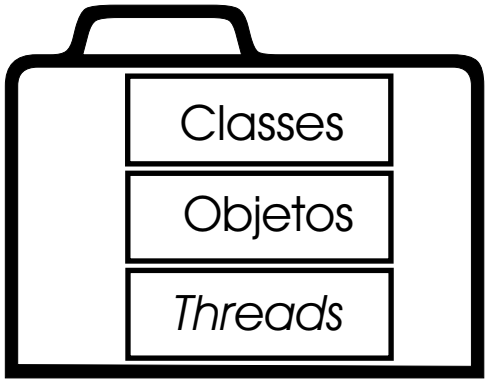


Figura 13: Ambiente μCODE - *Group*

realocação do código, ao *Group Handler* fica a responsabilidade pelo gerenciamento dos elementos recebidos em um *Group* - o que, em linhas gerais, envolve a extração e utilização das classes e objetos recebidos na criação de uma nova unidade de execução.

Durante o processo de reconstrução de um *Group*, o conteúdo é alocado em alguma porção da memória até que o *Group Handler* assuma a gerência desses elementos, esse local é conhecido pelo nome de *namespace*. No ambiente μCODE o controle dessas porções de memória fica a cargo do *ClassSpace*, que também é o responsável por evitar conflitos entre elementos oriundos de diferentes *Groups*. O tratamento de tais conflitos, fica a cargo do *μServer* que cria e associa uma instância diferente de um *ClassLoader* para cada *ClassSpace*

Tais conceitos se constituem no núcleo do ambiente μCODE, implementados através do pacote `mucode`, e também servem como base para a criação de abstrações de alto-nível, dentre as quais podem ser citadas as providas pelo pacote `mucode.abstractions`, que incluem por exemplo, uma abstração de agente móvel, o *MuAgent*.

As alterações introduzidas na versão segura do ambiente μCODE, objetivaram aplicar os conhecimentos relativos à assinatura de código, na criação de uma unidade de migração digitalmente assinada, o chamando *SignedGroup*, tornando possível alcançar o primeiro objetivo do mecanismo de segurança proposto para o ambiente μCODE, ou seja, criar a possibilidade de verificação e validação da origem de um *Group*.

A Figura 14 apresenta as principais alterações realizadas na unidade de migração do ambiente μCODE, onde se observa que, além dos elementos originalmente armazenados, em um *Group*, também estarão presentes a assinatura digital e a identificação do usuário da aplicação.

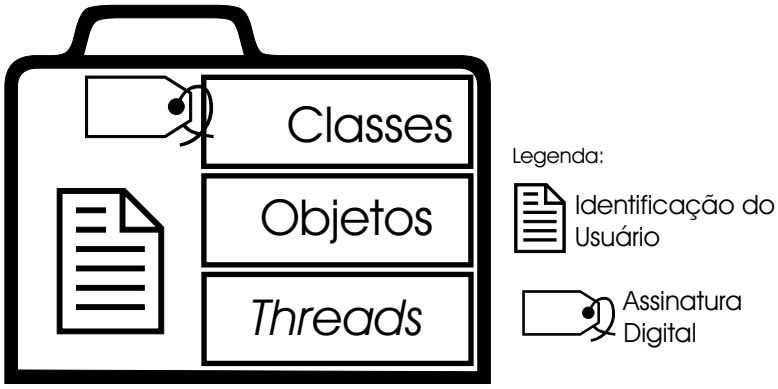


Figura 14: Ambiente μCODE Seguro- *Signed Group*

Na prática, o principal impacto causado pela criação dessa nova unidade de migração está na especialização da classe *Group* em uma classe chamada *Signed Group*, o que efetivamente passou a permitir a utilização do recurso de assinatura de código junto ao grupo. Assim, no momento da realização da transferência de um grupo assinado entre servidores *μServer*, será necessário que o usuário informe sua identificação e chave privada.

Dessa forma, um vetor de *bytes* (objeto do tipo `ByteArrayOutputStream`) é instanciado e preenchido com todo o conteúdo que formará o grupo, para posteriormente ser assinado com base nas informações passadas pelo usuário. Vale ressaltar, que esta assinatura engloba apenas as classes que irão compor um grupo, uma vez que a inclusão dos objetos tende a inviabilizar o mecanismo em função de que qualquer alteração em algum dos objetos causará divergências entre os valores computados no momento da verificação da autenticidade do grupo.

Tomemos como exemplo um programa que tenha por objetivo buscar um arquivo, determinado pelo usuário, em diversos servidores *μServer*. Após sua inicialização, esse programa cria um *SignedGroup* que será preenchido com três classes e um objeto, a assinatura do grupo será um valor computado a partir da chave privada do usuário e no próprio conteúdo do grupo. Após a migração e validação deste grupo, as classes e o objeto serão descompactados, e a busca pelo arquivo será realizada no servidor, o que se repetirá até que o arquivo seja encontrado. Neste momento, o grupo retornará à máquina do usuário.

Se a assinatura fosse aplicada a todo o conteúdo de um *SignedGroup*, e não somente às classes que o compõe, no momento em que este grupo fosse recebido por um servidor *μServer*, após encontrar o arquivo especificado, a aplicação apresentaria uma

divergência em seu comportamento. O grupo seria rejeitado durante a validação da assinatura digital, um vez que houve alteração em alguns dos elementos que compõe o grupo, neste caso o objeto, responsável por registrar a presença do arquivo.

Essa divergência pode ser justificada, em virtude da alteração no valor obtido durante a validação da assinatura, visto que, por ter sofrido uma alteração, mesmo que aparentemente ínfima, o valor computado sobre o *SignedGroup* já será divergente do valor inicialmente obtido.

A Figura 15 representa a migração de um grupo digitalmente assinado pelos servidores *µServer* de uma aplicação, assim, a partir da máquina "Cliente", um *SignedGroup* possui condições de migrar para os diversos servidores *µServer* que compõe uma aplicação, as setas na Figura 15 reforçam a não existência de uma ordem pré-estabelecida para visita dos servidores.

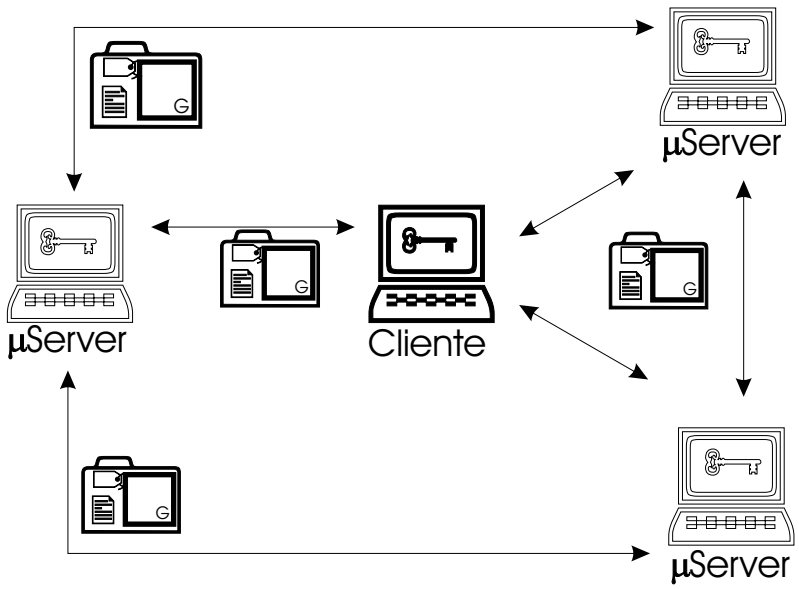


Figura 15: Ambiente µCode Seguro - Migração

O servidor *µServer*, ao receber um grupo assinado realiza uma consulta à chave pública do usuário identificado como emissor de tal grupo e uma vez verificada tal assinatura dá-se início à extração do conteúdo que forma o grupo. Entretanto, se houver divergência entre a assinatura do grupo e a chave pública do usuário, uma exceção será gerada e a execução encerrada. Se a verificação da assinatura for realizada com sucesso, os elementos que compõe o grupo serão alocados em um *ClassSpace*, cabendo ao *µClassLoader* controlar o posterior carregamento e execução das classes.

A versão segura do ambiente µCode, apresenta uma versão adaptada do *µClassLoader*,

denominado *SecureμClassLoader*, em que além da manipulação e gerenciamento das classes há também a possibilidade de aplicação de uma política de segurança a um grupo digitalmente assinado.

Tendo em vista o suporte oferecido pelo ambiente μCODE a diversos paradigmas de mobilidade de código, também se fez necessário a definição e a implementação de versões seguras para as demais abstrações oferecidas pelo ambiente, dentre as quais podemos destacar a criação da classe *SecureMuAgent*.

Contudo, destaca-se que o mecanismo aqui descrito é válido independentemente do paradigma utilizado na concepção do ambiente, visto que as alterações estão restritas à unidade de migração. Outro fator relevante é o papel desempenhado pelos servidores *μServer* que assumem uma postura passiva com relação às assinaturas do grupo, assim, realizada a primeira migração, quando o grupo é digitalmente assinado pelo usuário, estes servidores conservarão a assinatura durante as transferências subseqüentes.

Por fim, um *Group* também pode conter classes que são conhecidas pela denominação de ubíquas, ou seja, classes ligadas à API da linguagem Java ou ao próprio ambiente μCODE que estão presentes em todos os servidores *μServer* em um domínio. Essas classes também podem estar presentes em um *SignedGroup*, contudo as classes ubíquas não serão diretamente afetadas pelo mecanismo de assinatura do grupo, entretanto, em função da atribuição de permissões que permitirão controlar a execução de uma aplicação, será preciso garantir um conjunto de privilégios a tais classes de forma que a aplicação e o próprio ambiente μCODE tenham condições de serem executados.

4.3 Permissões

Introduzidas no mecanismo de segurança da linguagem Java a partir da versão 1.2 (Java 2), as permissões desempenham um importante papel na versão segura do ambiente μCODE, uma vez que caberá a elas disciplinar a forma como as aplicações são executadas, através da chamada Política de Segurança. Deste modo, além de garantir a origem de uma aplicação por meio da assinatura digital dos grupos, torna-se possível estabelecer diferentes níveis de acesso e utilização tanto para as informações, como para os recursos disponibilizados pelo sistema operacional.

A política de segurança foi um dos fatores que contribuiu para a introdução de

um grau maior de flexibilidade junto ao mecanismo de segurança da linguagem Java, tido até aquele momento como muito restrito em diversas situações. Assim, por meio da política de segurança criou-se a possibilidade da atribuição de permissões individuais que garantem, ou não, a execução de um código. Essas permissões estão baseadas nas seguintes características: quem assina o código, qual a origem ou ainda se o código foi digitalmente assinado, dessa forma, cria-se uma matriz que define o comportamento das aplicações especificando quais os recursos de sistema podem ser utilizados, de que maneira e sob quais circunstâncias, conforme mostra a Figura 16.

De acordo com a figura, está definido que todo código assinado pelo usuário *John Doe* possuirá condições para ler e escrever no diretório `/tmp`, assim como qualquer outra aplicação executada localmente. Por outro lado, os códigos oriundos da URL do Laboratório de Teste de Software, indepedentemente se são assinados ou não, poderão atender pedidos e realizar conexões com o endereço: `http://lost.fundanet.br`.

Figura 16: Matriz - Usuários X Permissões

Dessa forma, ao executar uma aplicação Java, as classes envolvidas são carregadas em um *ClassLoader*, que compõe a parte central da arquitetura de segurança da linguagem Java e cujo papel compreende tanto a eventual busca por algum bytecode solicitado pela máquina virtual, como o gerenciamento e relacionamento das classes armazenadas nos diferentes *namespaces* controlados. Vale ressaltar, que, durante o processo de alocação das classes em um *Class Loader*, estas passam em um primeiro momento pelo *Verifier*, sendo ele o responsável por realizar uma análise preliminar sobre a confiabilidade de um código, antes mesmo que o código tenha permissão para ser executado.

Se nenhum mecanismo de segurança for invocado, as classes de uma aplicação Java, depois de carregadas pelo *ClassLoader* serão consideradas confiáveis, o que implicará

na ausência de controle sobre as operações realizadas. Dessa forma, a aplicação possuirá condições plenas para realizar as mais diversas operações, como, por exemplo: operações com arquivos, acesso a propriedades de sistema, abertura de conexões de redes, instanciar novos processos, entre outros.

Porém, uma vez invocado o mecanismo de segurança, vem à cena a terceira parte que compõe o modelo de segurança da linguagem Java, o *Security Manager*. Recai sobre ele grande parte das funções desempenhadas por todo o modelo de segurança, sendo a ele atribuída a tarefa de manter em tempo de execução o registro de quem está autorizado a realizar quais operações consideradas potencialmente danosas. Em linhas gerais, a atuação do *Security Manager* pode ser resumida da seguinte maneira:

- Uma aplicação Java solicita à API a execução de uma operação considerada perigosa ao sistema.
- O *ClassLoader* consultará o *Security Manager* se tal operação deve ser permitida.
- Se não há permissão para tal operação, o *Security Manager* gera uma exceção (`SecurityException`) que é propagada à aplicação.
- Se houver permissão para realizar a operação, o *Security Manager* não retornará nenhuma exceção e a operação será realizada e a execução da aplicação seguirá normalmente.

A política de segurança atua em conjunto com o *Security Manager*, realizando o mapeamento a partir de uma identidade - assinatura digital e/ou origem do código - em um conjunto de permissões que garante a execução e o acesso a operações e recursos. Isso faz com que a identidade de um código seja confrontada com as entradas que formam a matriz de política de segurança, determinando quais permissões serão garantidas ao código de uma aplicação.

Na prática, a política de segurança é determinada em um arquivo de configuração escrito em formato ASCII, que possui extensão `.policy` e que pode ser criado por usuários ou administradores do sistema, sendo carregado junto à máquina virtual Java no momento de sua inicialização. A seguir encontra-se um exemplo de arquivo de políticas (Figura 17), onde são definidas um conjunto de permissões aplicadas a códigos assinados por *sysadmin* e oriundos do diretório */home/sysadmin* e seus subdiretórios. Também ficou definido que

todos tem garantidos o privilégio de leitura para o item *java.vendor* e por fim, códigos assinados por *Duke* possui tanto permissão para escrever como para ler no diretório */tmp/*.

```
Arquivo de Políticas: java.policy

grant signedBy "sysadmin", codeBase "file:/home/sysadmin/*" {
 permission java.security.SecurityPermission "Security.insertProvider.*";
 permission java.security.SecurityPermission "Security.removeProvider.*";
 permission java.security.SecurityPermission "Security.setProperty.*";
};

grant {
  permission java.util.PropertyPermission "java.vendor", "read";
};

grant signedBy "Duke" {
  permission java.io.FilePermission "/tmp/*", "read,write";
}
```

Figura 17: Exemplo: Arquivo de Políticas

No momento da inicialização de uma aplicação, é possível especificar tanto o uso do mecanismo de segurança como o arquivo com a política de segurança a ser utilizada, o que é feito por meio do flag `-D` que permitirá a definição de propriedades junto à máquina virtual, como é mostrado a seguir, onde além da chamada ao mecanismo de segurança é indicado o arquivo de políticas `java.policy`:

```
$java -classpath C:\mucode\mucode.jar;. -Djava.security.manager
 -Djava.security.policy=java.policy BounceLauncher localhost:2000 localhost
```

Observa-se que é feita a chamada à máquina virtual Java, indicando o caminho ao local onde μCODE está instalado, a seguir os parâmetros *-Djava.security.manager* e *-Djava.security.policy*, realizarão, respectivamente, a definição do mecanismo de segurança da linguagem e tambem sinalizarão com qual arquivo de políticas devera guiar a execução de tal aplicação.

O ambiente μCODE, por estar baseado na linguagem Java, preserva a essência do princípio de funcionamento até aqui descrito. Uma vez realizada a migração de um *Group*, este, ao chegar a um servidor *μServer*, terá suas classes manipuladas por uma versão especializada do *ClassLoader*, denominada *μClassLoader*. Apesar das peculiaridades pertinentes ao empacotamento, transferência e reconstrução de um grupo, o funcionamento do *μClassLoader* é análogo ao *ClassLoader*, assim, como acontece a qualquer aplicação Java quando o mecanismo de segurança da máquina virtual Java não é invocado, uma

aplicação do ambiente μCODE após carregada pelo *μClassLoader* está em condições de ser executada sem que haja controle sobre as ações e operações realizadas.

Por essa razão, apenas garantir a autenticidade de um grupo recebido por um servidor *μServer* torna-se insuficiente se o objetivo for à criação de uma versão segura do ambiente μCODE. O que leva a necessidade de também estabelecer níveis de acesso à execução das aplicações, seja em termos de informações como na utilização de operações que possam a vir causar danos ao sistema ou ao ambiente do usuário. Em razão deste fato, a consolidação de uma versão segura ao ambiente μCODE, implica em vincular políticas de segurança junto à execução das aplicações baseadas no ambiente μCODE.

Dentre os reflexos dessa situação, está a adequação do *μClassLoader*, realizada conforme a descrição a seguir.

4.4 Adequação do *μClassLoader*

Na versão segura do ambiente μCODE, existe a transferência de versões digitalmente assinadas da unidade de migração, o *Signed Group*, dessa forma, antes mesmo que ocorra a análise preliminar realizada pelo *Verifier*, será realizada a verificação da autenticidade do grupo recebido. Destaca-se que a alocação e gerenciamento das classes que compõem o grupo, e conseqüentemente sua execução, está na pendência do valor computado entre a assinatura digital e a chave pública armazenada no servidor *μServer* e que está associada à identificação do *Signed Group* recebido. Havendo divergência no valor calculado, a aplicação terá sua execução sumariamente interrompida.

Neste ponto, assumindo o carregamento das classes transferidas através de um *Signed Group*, se faz necessário associar tais classes a uma política de segurança. Isso demanda alterações estruturais para adequação do *μClassLoader*, que não mais especializará a classe *Class Loader*, passando a ser herdeira da classe `SecureClassLoader` (API `java.security`).

Esta modificação introduz maior flexibilidade à definição das classes tratadas por um *μClassLoader*, a partir de agora, em função da associação de um *μClassLoader* a um único *ClassSpace* e da associação deste a um único usuário, definido na forma de uma URL, torna-se possível criar uma associação entre o *μClassLoader* e a URL correspondente a um usuário, forçando assim, que o *ClassLoader* atribua às permissões definidas àquela

URL.

O trecho de código abaixo (Figura 18), extraído do método `loadClass` da classe *μClassLoader*, destaca o uso do método `defineClass` (linha 23), vale mencionar que o parâmetro `codeSource` (cs), oferece ao usuário a possibilidade de informar sua identificação, por meio de qualquer URL válido, como por exemplo: `mailto:leonardo@fundanet.br`, `file://rodrigo`, `http://lost.fundanet.br`, entre outros.

```
1  public synchronized Class loadClass(String name, boolean resolve)
2    throws ClassNotFoundException {
3    Class c = null;
4    String id = null;
5    byte[] byteCode = null;
6
7    if (server == null || server.isUbiquitous(name)) {
8      c = findSystemClass(name);
9      if (c == null) throw new ClassNotFoundException();
10   } else {
11     // If not, let's check whether it's already in our class space.
12     c = classSpace.getClass(name);
13     if (c != null) server.D(name + " class retrieved " +
14                             "from the thread private class space.");
15     else {
16       // If not, check whether there's the bytecode ready for it.
17       try {
18         byteCode = classSpace.getClassByteCode(name);
19       } catch (IOException e) {
20         throw new ClassNotFoundException(e.toString());
21       }
22       if (byteCode != null) {
23         c = defineClass(name, byteCode, 0, byteCode.length, cs);
24         classSpace.putClass(name, c);
25         server.D(name + " class retrieved from the group's stream.");
26       } else {
27         // If not, check whether it's in the shared class space
28
```

Figura 18: Classe *μClassLoader* - método `loadClass`.

Dessa forma, da mesma maneira como acontece com as aplicações Java, no momento que as classes estiverem sendo carregadas pelo *μClassLoader*, serão associadas a um arquivo de políticas que por sua vez, em conjunto com o *Security Manager* irão atuar na garantia dos privilégios que os usuários possuem para executar as operações.

Assim, no momento da criação do *Signed Group* será pedido que o usuário informe sua chave privada e sua identificação - por meio de uma URL válida - e o grupo será criado e assinado. Após o recebimento pelo servidor de destino, realizada as verificações necessárias, o conteúdo do grupo será carregado pelo *μClassLoader* e associado a um arquivo de políticas, que utilizará o parâmetro `codeSource` para confrontar as permissões atribuídas à referida identificação por meio da cláusula `codeBase` no arquivo de políticas.

A Figura 19 mostra um arquivo de política utilizado no contexto de uma aplicação baseada na versão segura do ambiente μCode, onde é possível perceber na última porção

do arquivo `java.policy` que os códigos identificados pela URL "file://rodrigo", possuem respectivamente as seguintes permissões para ler o arquivo `bla.txt` armazenado no diretório `/home/rfa` além de privilégios para criar e resolver conexões do tipo `socket` com o *localhost*.

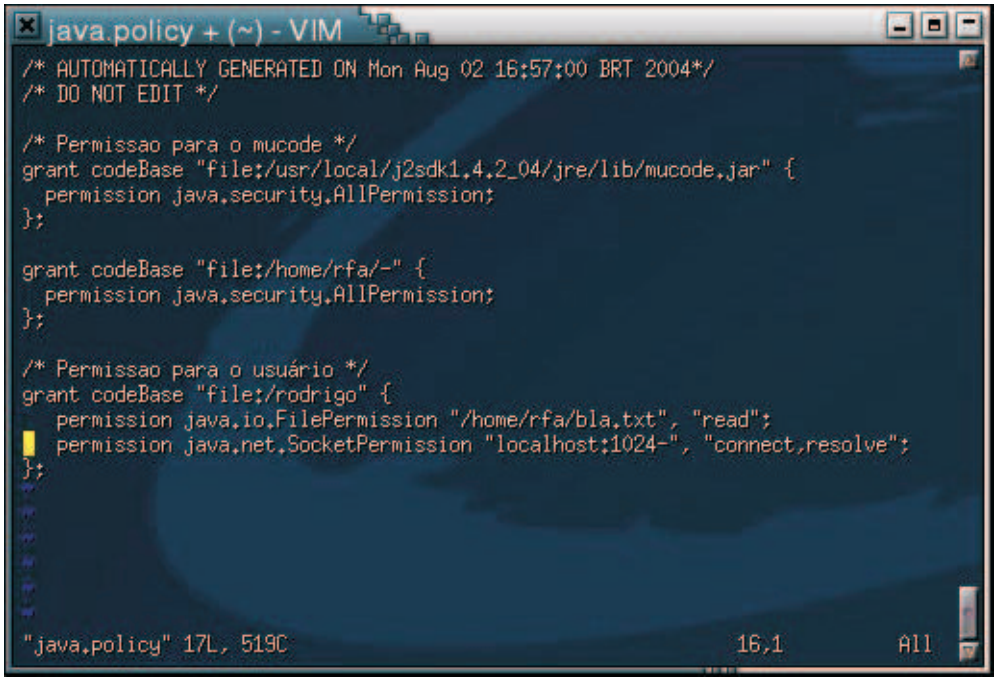


Figura 19: Ambiente μCode Seguro - Arquivo de Políticas.

A primeira porção do arquivo `java.policy` (Figura 19), apresenta o conjunto de permissões que são necessárias para que as classes denominadas ubíquas tenham condições de serem executadas, ou seja, para que as classes que estão presentes e disponíveis a todos os servidores *μServer*, ou que façam parte do ambiente μCode, tenham garantido os privilégios exigidos para sua execução. A seguir estão as permissões necessárias para que a aplicação tenha condições de executar.

Em conseqüência da flexibilidade oferecida pelo ambiente μCode, além da criação da versão segura do *Group*, fez-se necessário também implementar versões seguras para as demais abstrações oferecidas pelo ambiente, que formam o pacote `mucode.abstractions`: **copyThread**, **spawnThread**, **shipClasses** e **fetchClasses**.

Tais primitivas permitem a realocação de classes e objetos do tipo *thread*, criando uma arquitetura para exploração da mobilidade de código sem necessariamente criar um agente móvel. A implementação do paradigma de agentes móveis cabe a classe `MuAgent`, e consequentemete sua versão segura a classe `SecureMuAgent`, que definem tanto o com-

portamento do agente móvel como o comportamento do *Group Handler* correspondente.

4.5 Considerações Finais

As alterações descritas representam as principais alterações realizadas para a construção de uma versão segura do ambiente μCODE. Tais modificações permitem a verificação e controle das ações, operações e recursos manipulados por uma aplicação baseada em código móvel, uma característica até então inexistente no ambiente em questão.

A completa implementação dessa versão segura, exigiu, ainda, a realização de pequenas alterações, de menor relevância, em sua maioria, para o adequado funcionamento do ambiente. Dentre as alterações realizadas, vale ressaltar a criação de versões segura para as abstrações oferecidas pelo ambiente μCODE, criando, por exemplo, o `SecureMuAgent`, preservando dessa forma um dos principais diferenciais do ambiente em relação aos seus pares, que a flexibilidade oferecida aos programadores e desenvolvedores de possuírem liberdade de escolha do paradigma a ser utilizado no momento da implementação de suas aplicações.

5 *Experimentos Realizados*

Visando demonstrar a aplicação do mecanismo de segurança implementado na versão segura do ambiente μCODE, este capítulo apresenta exemplos que exploram as funcionalidades introduzidas. Esses exemplos procuram demonstrar o uso de tal mecanismo de segurança, reforçando a idéia da sua aplicabilidade independentemente do paradigma utilizado, assim, são utilizados dois exemplos baseados no paradigma de agentes móveis sendo que, o primeiro exemplo, demonstra o funcionamento da migração de código, enquanto que no segundo exemplo é feita a simulação de um jogo conhecido em algumas regiões do Brasil pelo nome de Bozó, enquanto que o segundo, demonstra a utilização da abstração segura para agentes móvei

Vale ressaltar que ambos os exemplos foram construídos antes da implementação do mecanismo de segurança, o que reforça a compatibilidade entre a versão original e a versão segura do ambiente μCODE.

5.1 Preparação do Ambiente

Para a execução dos exemplos aqui apresentados, além da instalação do ambiente μCODE, foi necessária a criação dos arquivos de políticas, além das chaves: pública e privada. Os arquivos com as políticas de segurança utilizados pelo servidor *μServer* e pelo cliente durante os experimentos, são apresentados a seguir (Figuras 20 e 21). A diferença entre os arquivos, está no fato de que, em função da inicialização da aplicação, o arquivo de políticas alocado na máquina do usuário, deverá posuir permissões suficientes que garantam a execução da aplicação, conforme pode ser visualizado através do segundo bloco de código mostrado na Figura 21.

Dessa forma, a máquina do usuário desempenha um duplo papel, a princípio

como Cliente - em função da inicialização da aplicação. E, posteriormente, como também pertence ao domínio da aplicação, essa mesma máquina tem a possibilidade de atuar como um servidor *μServer*.

```
/* Arquivo de Políticas */

/* Permissao para o mucode */
grant codeBase "file://usr/localj2sdk1.4.2_04/jre/lib/mucode.jar"{
   permission java.security.AllPermission;
};

/* Permissao para o usuario */
grant codeBase "file://rodrigo"{
   permission java.net.SocketPermission "localhost:1024-","connect,resolve";
   permission java.io.FilePermission "/home/rfa/bla.txt","read,write";
};
```

Figura 20: Política de Segurança - Servidor

Exceto pelas permissões específicas, relacionadas a inicialização da aplicação, não há maiores diferenças entre os arquivos. Sendo que, na primeira porção dos arquivos, em função das classes ubíquas, os arquivos mostram o conjunto de permissões atribuídas ao ambiente μCode, necessárias para que o ambiente tenha condições de executar, do contrário nem mesmo o ambiente possuirá privilégios suficientes que garantam sua execução. Ao final, estão definidas as permissões aplicadas em nível individual, sendo para tanto, atribuídos os privilégios pertinentes a cada usuário. Neste caso, nota-se a presença de permissões associadas a um usuário denominado "rodrigo", que possui possui privilégios suficientes para realização de várias tarefas, como por exemplo, operações com arquivos.

Vale ressaltar que o arquivo com a política de segurança, deverá estar presente nos diversos servidores *μServer* que compõe o domínio da aplicação.

A Figura 22, representa a alocação dos arquivos de políticas no contexto de uma aplicação para o ambiente μCode Seguro, dessa forma, o servidor MS_a, como todos os outros servidores presentes naquele domínio de aplicação, deverá possuir o arquivo de políticas `java-server.policy`. A máquina do usuário *Cliente*, apresenta também o mesmo arquivo de políticas, onde, além das permissões comuns a todos os servidores, estarão também garantidos os privilégios necessários a execução da aplicação. Que deverá executar, até o momento de migrar a outro servidor *μServer*, quando seus dados serão empacotados na unidade de migração do ambiente μCode, assinados e transferidos a outro servidor *μServer*, que no momento adequado também empacotará e realizará a transferência dos dados, até que este chegue novamente a máquina de origem, ou seja, a

```
/* Arquivo de Políticas */

/* Permissao para o mucode */
grant codeBase "file://usr/localj2sdk1.4.2_04/jre/lib/mucode.jar"{
   permission java.security.AllPermission;
};

/* Permissao para a aplicacao */
grant codeBase "file://home/rfa/-"{
   permission java.security.AllPermission;
};

/* Permissao para o usuario */
grant codeBase "file://rodrigo"{
   permission java.net.SocketPermission "localhost:1024-","connect,resolve";
   permission java.io.FilePermission "/home/rfa/bla.txt","read,write";
};
```

Figura 21: Política de Segurança - Máquina do Cliente

máquina que iniciou o processo.

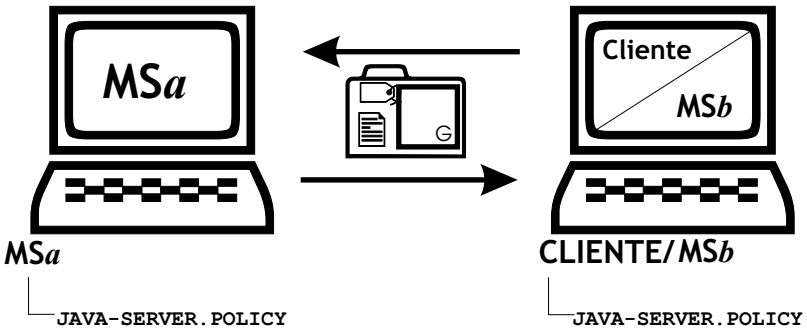


Figura 22: Alocação dos arquivos de políticas de segurança

A instalação do ambiente foi realizada conforme as orientações disponíveis no website do projeto μCODE, sendo que posteriormente foi realizada a substituição dos arquivos `mucode.jar` e arquivos-fonte presentes no diretório */mucode/source*, pelos arquivos gerados no decorrer deste projeto. As chaves utilizadas na execução do experimentos, foram geradas por meio do programa `GenKey.java` (anexo A - p. 91), sendo que nestes exemplos, a chave privada é armazenada em um arquivo identificado pelo nome do usuário com a extensão `privkey`, enquanto que a chave pública, por sua vez, adotará a extensão `pubkey`.

Os experimentos realizados se valem da utilização de duas ou mais instâncias da máquina virtual Java, sendo executadas simultaneamente na mesma máquina ou em diferentes máquinas de uma rede. A principal diferença entre elas é que em se tratando de máquinas diferentes de uma rede, serão utilizados os seus endereços na rede ao passo que, quando executadas na mesma máquina, serão atribuídos diferentes números para as

portas, por exemplo: `localhost`, `localhost:2000` e `localhost:3000`.

5.2 *Experimento A*: Agente Móvel

Uma das principais características do ambiente μCODE é a flexibilidade dada aos desenvolvedores, que tanto podem se valer das abstrações oferecidas pelo próprio ambiente, como implementar seus próprios programas de acordo com suas próprias noções de mobilidade de código, ou conveniência. Este experimento, através da classe `MuAgent`, implementa um agente que migra continuamente entre dois servidores *μServer*.

Funcionamento: Este exemplo demonstra a funcionalidade da classe `MuAgent`, pertencente ao pacote `mucode.abstractions`, que por meio da criação de um agente móvel continuamente migra de um servidor *μServer* para outro. Quatro classes compõem esse exemplo: `Bouncer`, `BounceLauncher`, `AbstractBouncer` e `Performer`, sendo que, a classe `Bouncer` é uma instância da classe abstrata `AbstractBouncer` e implementa a interface `Performer`. Todas as classes são automaticamente empacotadas com o agente e remetidas ao destino, através do método `go()` pertencente à classe `MuAgent`.

Na prática, o primeiro passo para a execução do experimento envolve a abertura de dois terminais, ou *shell*, do sistema operacional sendo que em um deles será executado um servidor *μServer* e no outro será inicializado o agente móvel. A inicialização do servidor, neste caso na porta 2000, é feita pela seguinte linha de comando:

```
$>java mucode.util.Launcher -port 2000
```

O agente móvel deverá ser inicializado em um outro terminal, através da linha de comando abaixo, vale ressaltar que os dois parâmetros exigidos são: os endereços dos servidores que o agente deverá migrar continuamente.

```
/mucode/examples/MobileAgent$>java BounceLauncher localhost:2000 localhost
```

A partir dai, será possível visualizar a execução da aplicação, por meio da migração do agente móvel entre os dois servidores.

Experimento: Com base no exemplo apresentado, o mesmo exemplo será agora executado na versão segura do ambiente μCODE, em especial dá-se destaque as três situações

que podem influenciar diretamente a execução da aplicação, sendo elas:

- Execução com privilégios garantidos;
- Execução com ausência de privilégios;
- Divergência entre as chaves pública e privada.

Na primeira situação, consideramos um usuário *confiável* executando o programa, apesar de inicialmente a principal diferença entre a versão segura do ambiente μCODE e a versão original parecer resumida ao uso do flag `-Djava.security.policy`, entende-se que anteriormente houve o estabelecimento de uma relacão de confiança entre as entidades envolvidas na execução da aplicação, de modo que a chave pública do usuário em questão já se encontra acessível ao servidor *μServer*.

A Figura 23, apresenta o terminal do sistema, ou *shell*, onde é feita a inicialização do servidor na porta 2000. Através dos parâmetros informados também é possível perceber o uso do arquivo de política de segurança chamado *java-server.policy*.

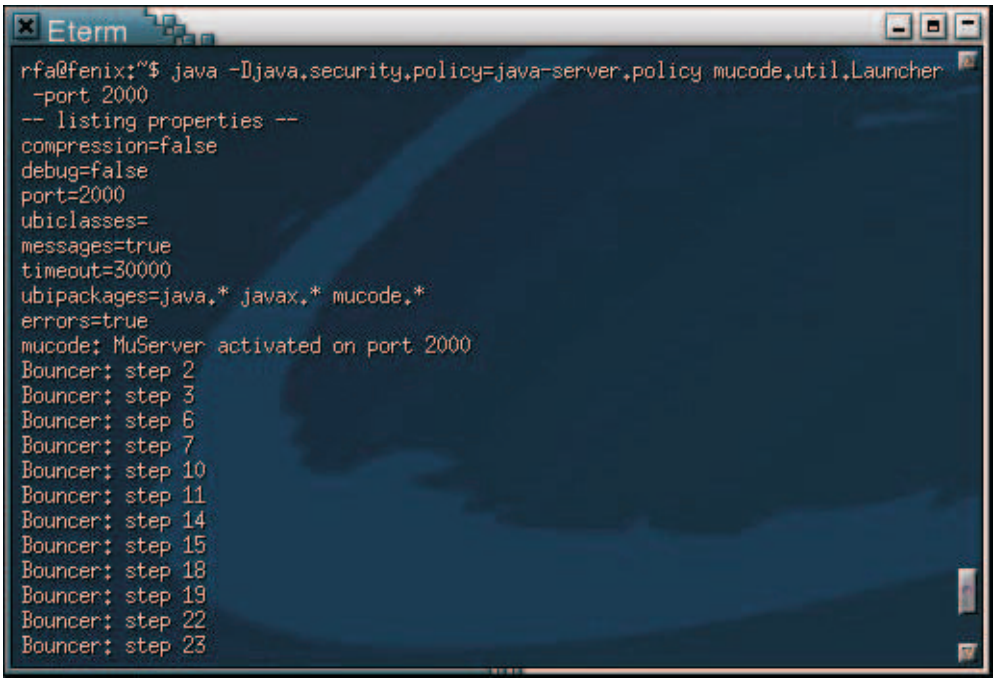


Figura 23: Execução - Terminal *μServer*

No terminal do sistema referente ao usuário (Figura 24), é feita a inicialização da aplicação, que nesta situação é responsável pela instanciação do agente que irá migrar entre dois servidores, vale ressaltar, que até este momento, só há um servidor *μServer* em

execução (na porta 2000). Por essa razão, a inicialização da aplicação, também implicará na instalação de um servidor *μServer* na máquina do usuário, pela Figura 24 observa-se que o servidor foi instalado na porta 1968.

Neste exemplo, o agente móvel é uma instância da classe abstrata `AbstractBouncer`, que por sua vez descende da classe `SecureMuAgent`, uma versão especializada da classe `MuAgent`, originalmente provida pelo ambiente μCODE. Trabalhar com versões assinadas da unidade de migração, o `SignedGroup`, é o principal diferencial entre a classe `MuAgent` e sua versão segura. A seguir é apresentado o trecho de código que define a classe abstrata `AbstractBouncer`:

```
import mucode.abstractions.*;
import mucode.*;
public abstract class AbstractBouncer extends SecureMuAgent {
  int step = 0;
  int lastVisited = 1;
  String[] hosts = new String[2];
  public AbstractBouncer() { super(); }
  public AbstractBouncer(String host1, String host2, MuServer aServer) {
    super(aServer);
    hosts[0] = host1;
    hosts[1] = host2;
  }
}
```

Por ser um usuário confiável, possuidor de um conjunto de permissões que garantem os privilégios suficientes para a execução, o programa será executado sem nenhuma restrição (Figura 25).

Neste exemplo, a identificação do usuário é relizada através do método `run()`, responsável por informar ao método `go()`, pertencente a classe `SecureMuAgent`, a identificação do usuário e sua chave privada, o que possibilitará a assinatura do *SignedGroup*.

```
public class Bouncer extends AbstractBouncer implements Performer {
  public Bouncer(String host1, String host2, MuServer aServer) {
     super(host1, host2, aServer);
```

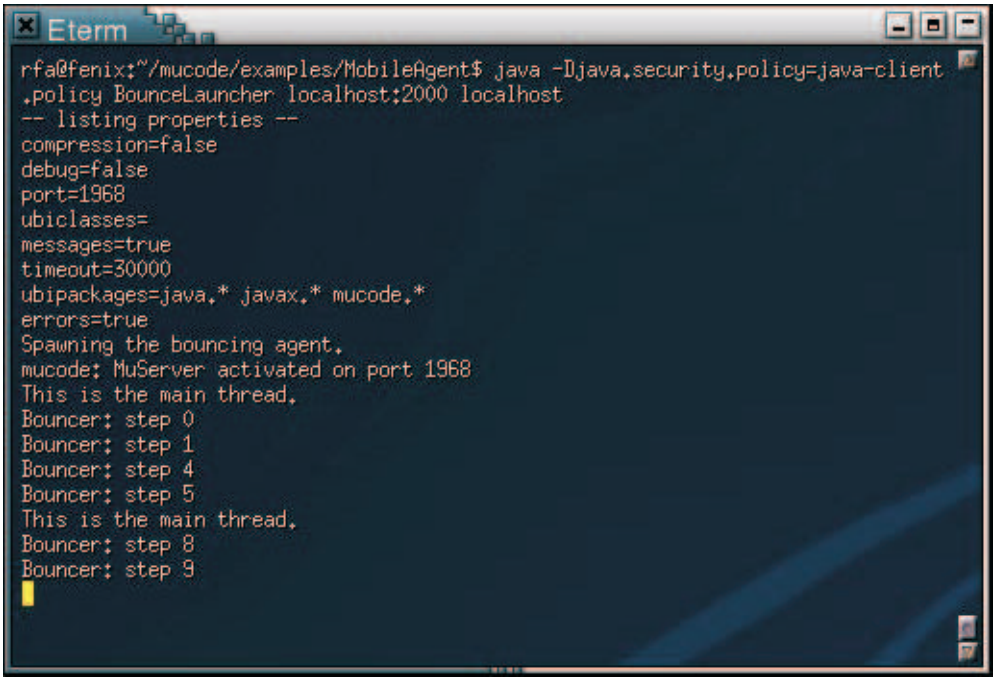


Figura 24: Execução - Terminal do Cliente - Inicialização da Aplicação

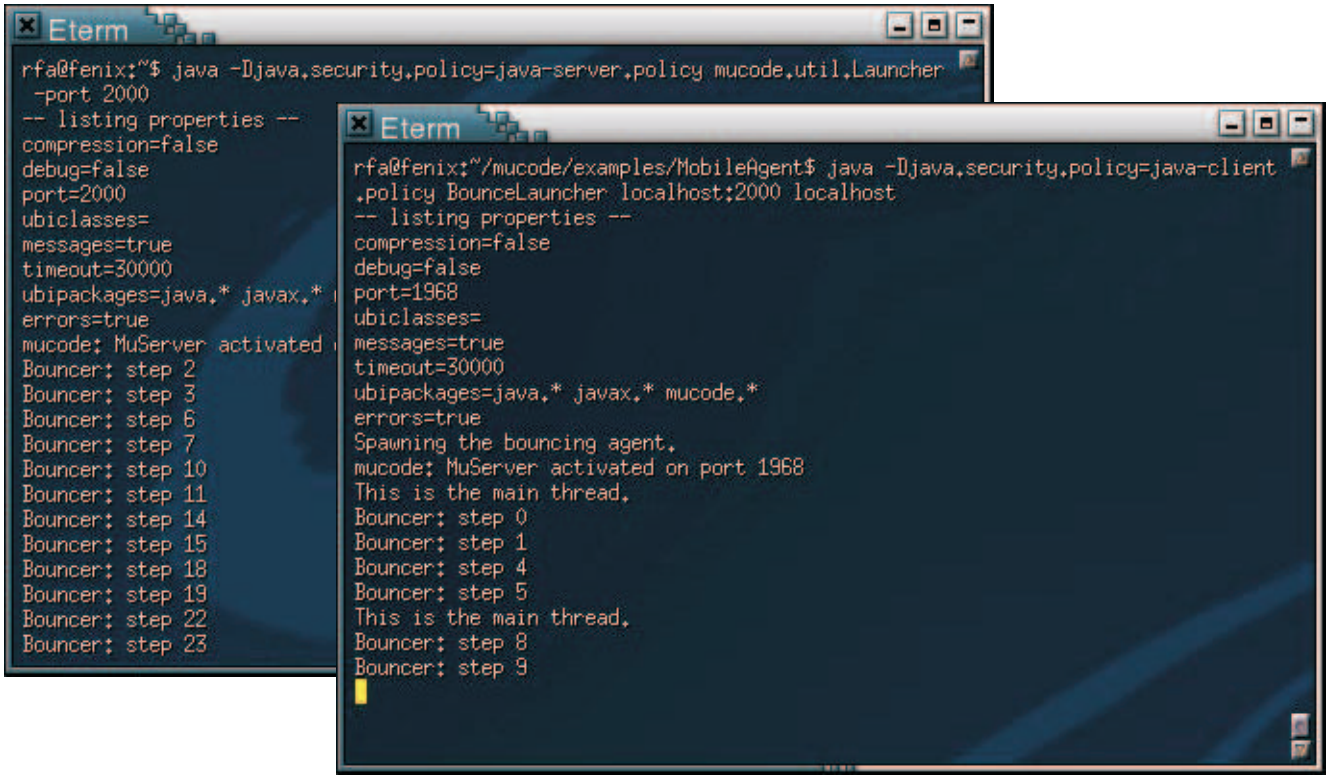


Figura 25: Execução - Experimento A: Agentes Móveis

```
}
public Bouncer() { super(); }
public void run() {
```

```
    for (;;) {
      try {
for (int i = 0; i < 2; i++, step++){
          perform();
          Thread.sleep(500);
  Thread.yield();
}
        if (lastVisited == 0) lastVisited = 1;
        else lastVisited = 0;
        go(hosts[lastVisited], "file:/rodrigo", "mucode.privkey");
      } catch (java.io.IOException e) { e.printStackTrace();
      } catch (ClassNotFoundException e) { e.printStackTrace();
      }
    }
  }

  public void perform() {
    System.out.println("Bouncer: step " + step);
  }
}
```

A validação da assinatura de um *SignedGroup*, realizada após a migração, é feita com base na chave pública do usuário. Para tanto, o servidor *μServer*, com base na identificação do usuário, realiza uma busca em um arquivo onde estão relacionados a identificação e o nome da chave pública de todos os usuários com os quais há relações de confiança.

Contudo, duas outras situações podem ser consideradas. Na primeira delas, considera-se a utilização do programa por um usuário que não possui privilégios suficientes para execução da aplicação, o que poderia acontecer caso a linha que garante o privilégio para que o cliente/usuário crie conexões via *socket* (`SocketPermission`) fosse retirada do arquivo de políticas. A Figura 26 mostra o terminal do servidor para a situação acima descrita.

Neste caso, a exceção é gerada em função da falta de privilégios que possibilitem ao agente realizar a migração de volta à máquina do cliente.

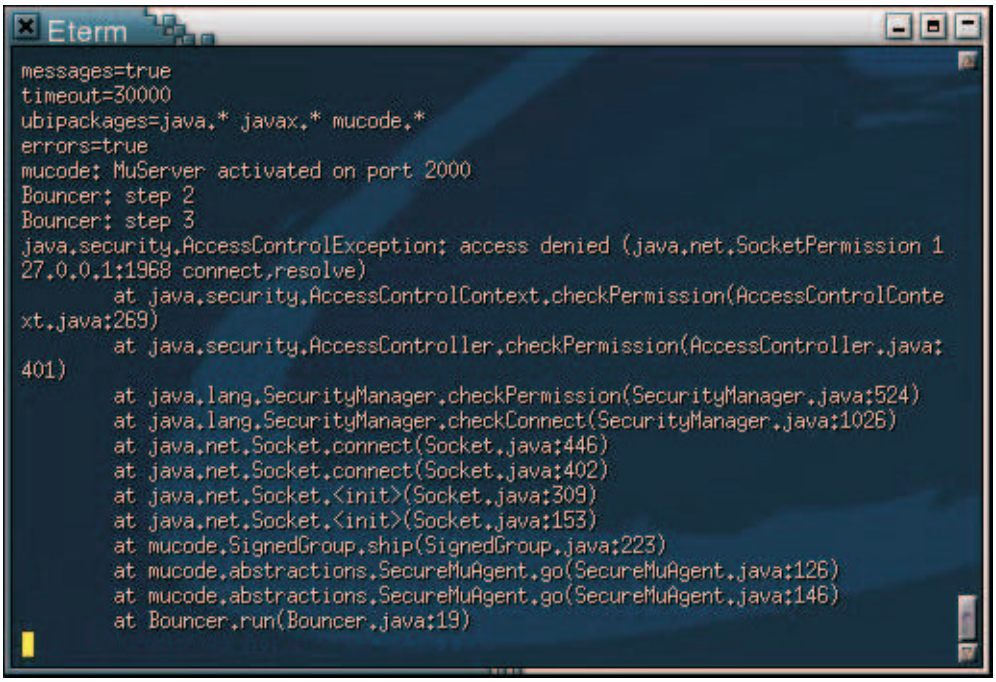


Figura 26: Execução - Terminal *μServer* - Violação da Política de Segurança

Por fim, podem ocorrer situações em que haja discrepância no valor computado entre as chaves pública e privada, seja por uma desatenção por parte do usuário, seja em função de uma intenção obscura. Independentemente do cenário, tal situação ocorre pela impossibilidade de uso de uma chave pública na validação de uma assinatura realizada; Cabe à chave pública, com base na assinatura digital recebida, verificar a autenticidade da chave privada utilizada pelo emissor. A Figura 27 mostra o terminal do servidor, para a situação descrita.

Em função da não validação do *SignedGroup*, o grupo não será manipulado pelo servidor *μServer* sendo uma exceção gerada.

5.3 *Experimento B*: Jogo de Bozó

Neste experimento tomaremos como base um programa escrito em linguagem Java que simula um jogo de dados muito popular em algumas regiões do Brasil, o Jogo de Bozó. Neste jogo, também conhecido como General, os participantes têm como objetivo preencher as treze posições do placar, sendo que cada posição garante uma certa quantidade de pontos.

A quantidade de pontos em alguns casos é fixa, dessa forma um *full-hand* sempre

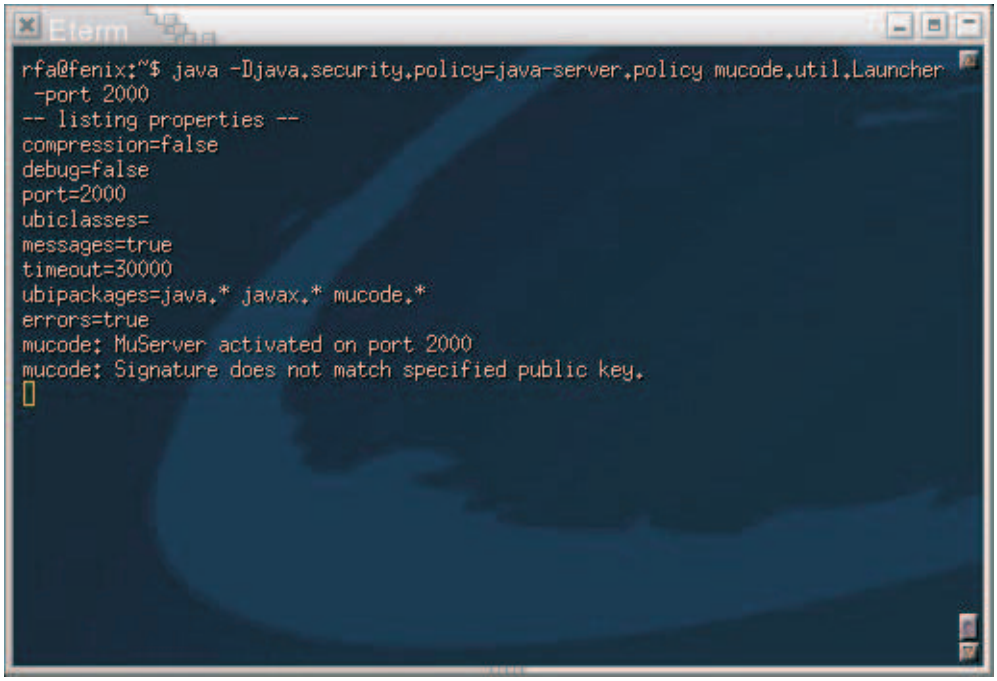


Figura 27: Execução - Terminal *μServer* - Violação da Política

garantirá 15 pontos ao jogador, toda vez que houver um conjunto formado por três dados com mesma face e um par com outra face, não importando nessa situação o valor presente na face dos dados. Em outros casos, os pontos obtidos por um jogador, será dependente dos valores obtidos nos dados. Por exemplo, a posição "seis" garante ao participante o número de dados com face seis, multiplicado por seis.

São utilizados cinco dados e cada jogador tem direito a três jogadas por rodada. Ao final de treze rodadas, todas as posições do placar deverão estar preenchidas, e os pontos alcançados em cada posição do placar deverão ser então totalizados de modo a fornecer o total geral de pontos de cada participante.

Funcionamento: Este programa, originalmente escrito como uma aplicação Java foi adaptado ao uso de agentes móveis[1]. Neste cenário, cada jogador é representado por um agente móvel, havendo, também, um servidor responsável por realizar e controlar as jogadas.

Dessa forma, a partir da máquina de cada jogador, um agente deve ser inicializado e enviado até o servidor com a intenção de dar início a um novo jogo. Cabe a esse servidor gerenciar o jogo, prover os dados, controlar o número de jogadas e rodadas, além de montar os placares com os pontos.

[1] Adaptação ao ambiente μCode realizada pelo professor Dr. Márcio Eduardo Delamaro.

Ao final do jogo, o agente retorna à máquina do cliente e apresenta o placar, bem como as jogadas e as escolhas realizadas.

Experimento: A seguir é apresentada a execução de uma aplicação que simula o jogo de Bozó, na versão segura do ambiente μCODE. Em um primeiro momento, é feita a execução dessa aplicação assumindo um cenário adequado, ou seja, uma situação em que um usuário confiável possui permissão para executar a aplicação.

O primeiro passo para execução deste exemplo, é a inicialização do servidor *μServer*, que, conforme foi dito anteriormente, será o responsável pelo controle do jogo. A Figura 28 apresenta o terminal com a inicialização do servidor. Vale destacar o uso de dois parâmetros informados na linha de comando:

- `-cp mucode.jar` - Que indicará ao interpretador da linguagem Java para adicionar o arquivo `mucode.jar` criado ao longo deste projeto e onde constam as alterações realizadas visando a criação da versão segura do ambiente μCODE junto ao *classpath* do sistema.
- `-Djava.security.manager=java-server.policy` - Através do flag `-D` é informado qual o arquivo de políticas de segurança que deverá ser utilizado, neste exemplo o arquivo `java-server.policy`.

Estando o servidor ativo, dá-se inicio ao jogo através da inicialização do agente móvel que irá até o servidor buscar as informações sobre o jogo, o que poderia ser feito através da linha de comando. (Figura 29):

```
$java -cp mucode.jar;. -Djava.security.policy=java-cliente.policy
 client.BozoClient2 127.0.0.1:1968
```

Através das Figuras 29 e 30 tem-se uma visão da execução da aplicação, sendo que na primeira figura é possível observar o início da mensagem trazida pelo agente, onde é apresentado o placar do jogo. A segunda figura, destaca o final da mensagem, onde é possivel visualizar as escolhas realizadas pelo servidor *μServer* nas últimas rodadas.

Se o objetivo fosse restringir as ações passíveis de execução por um usuário, bastaria aplicar-lhe as restrições desejadas através do arquivo de políticas presente no

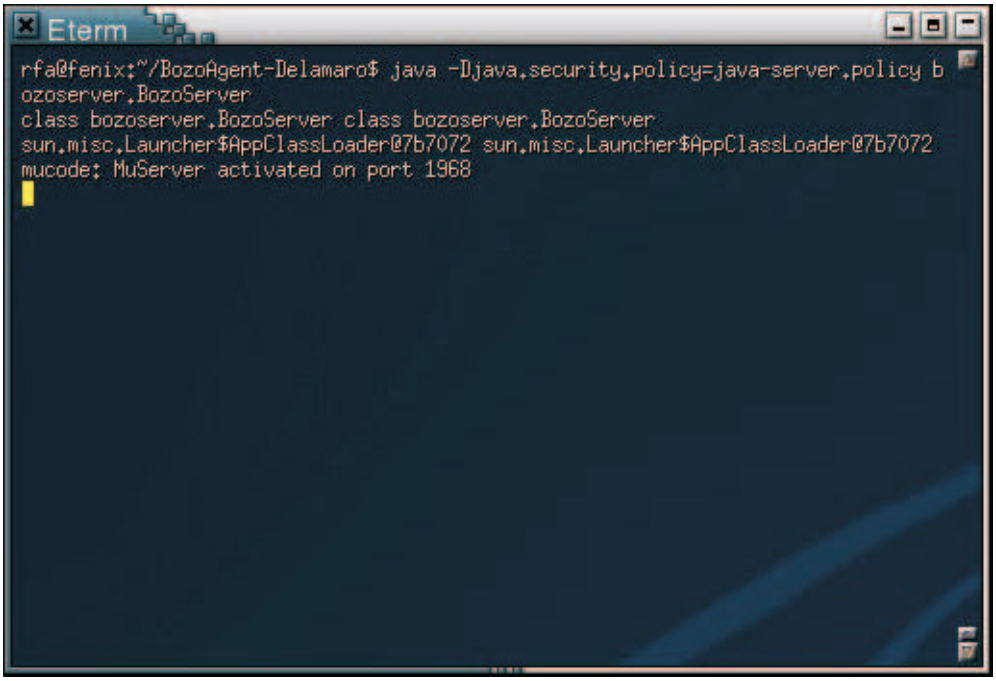


Figura 28: Execução - Terminal *µServer* - Inicialização do Jogo de Bozó

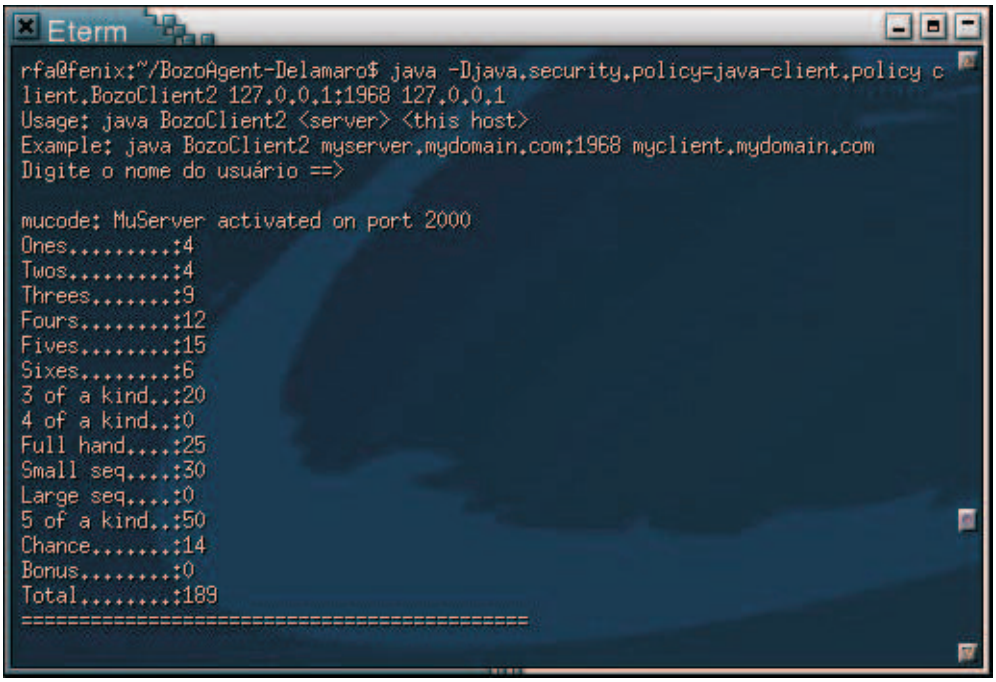


Figura 29: Execução - Terminal do Jogador - Inicialização do Agente

servidor *µServer*, uma vez alterado o arquivo, este deverá ser redistribuído a todos os servidores que compõe o domínio.

Se, por exemplo, a linha que garante ao usuário os privilégios para abertura e resolução de conexões do tipo *socket*, por alguma razão fosse suprimida, ou comentada,

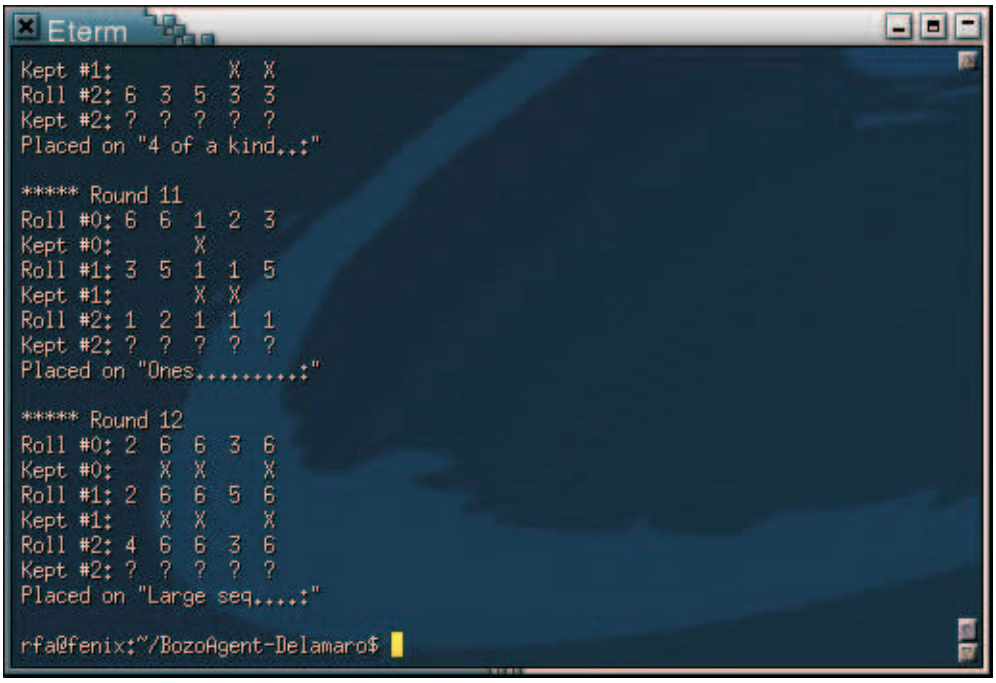


Figura 30: Execução - Terminal do Jogador

a aplicação seria levada a um estado de inconsistência durante sua execução. Visto que, após inicialização e migração do agente móvel ao servidor BozoServer, este verificaria que o usuário em questão não possui permissões que o permitam realizar a abertura de conexões na rede, o que impossibilitaria o retorno do agente móvel à máquina do cliente (Figura 31).

5.4 Considerações Finais

Os experimentos aqui apresentados, apesar de não abrangerem toda a gama de possibilidades de uso, procuram oferecer uma melhor compreensão sobre o funcionamento do mecanismo de segurança implementado na versão segura do ambiente μCODE. Em termos mais específicos, esta versão do ambiente μCODE Seguro, permite proteger os servidores *μServer* do ataque dos chamados *códigos maliciosos*, o que por vezes fomentou diversas discussões, sobre os aspectos práticos da implementação, dentre os quais destacamos:

- Compatibilidade entre versões. Não foi objetivo do projeto criar um novo ambiente, e sim, adequá-lo através da implementação de recursos originalmente ausentes.

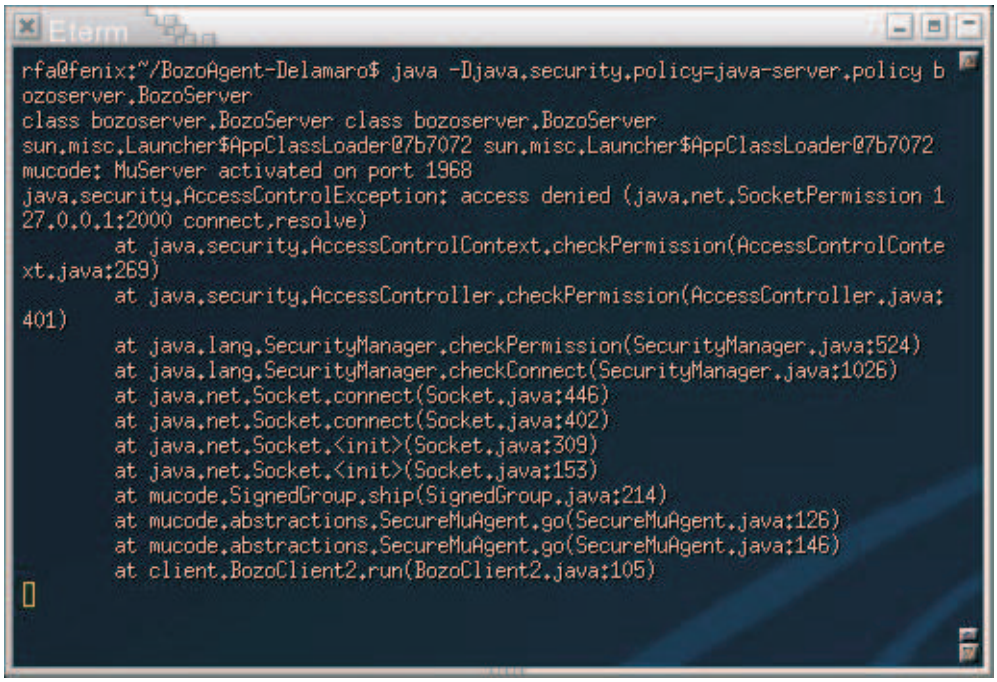


Figura 31: Execução - Terminal *μServer* - Falta de Privilégios

- A necessidade de atribuir a cada grupo assinado recebido, um *ClassLoader* específico, como forma de aplicar corretamente as restrições impostas pela política de segurança, evitando a ocorrência de conflito entre as permissões atribuídas aos usuários.

- A assinatura digital dos elementos que compõem um *SignedGroup*, apesar de inicialmente haver a intenção de assinar digitalmente todo o conteúdo que formaria a unidade de migração assinada; a prática, até o momento, mostrou-se inviável em razão das alterações sofridas pelos elementos do grupo.

- A definição clara do papel atribuído aos servidores *μServer*, que assumem uma postura de passividade frente ao mecanismo das assinatura dos *SignedGroup*. Por essa razão, realizada a primeira migração, os servidores apenas dão prosseguimento ao processo de migração sem que ocorra nova assinatura daquele grupo.

- Adequação das abstrações originalmente providas pelo ambiente, com o intuito de manter a compatibilidade entre as aplicações.

Em seu atual estágio de desenvolvimento, a versão segura do ambiente μCode, já oferece aos administradores, meios para controlar e verificar a forma como as aplicações executam neste ambiente. Programadores e desenvolvedores, também podem, optar pelo

uso das abstrações originalmente providas, ou então utilizar as versões seguras criadas tais abstrações. Apesar de não encerrar o assunto, o primeiro passo já foi dado na intenção do oferecimento de uma versão segura do ambiente μCODE, no sentido mais amplo da palavra. Contudo, a fim de abranger o maior número possível de situações, outros experimentos deverão ser realizados com essa versão, visando explorar as diversas nuances apresentadas pelo μCODE Seguro.

6 Conclusões

Com o objetivo de criar um mecanismo de segurança para o ambiente μCODE, o presente trabalhou abordou aspectos práticos e teóricos a respeito do uso da mobilidade de código no desenvolvimento de sistemas distribuídos, sendo que, em função de estarem em contato com um ambiente tão heterogeneo, são exigidas características como: flexibilidade e configurabilidade.

A ausência de tais mecanismos de segurança tem sido atribuída como uma das principais razões que contribuem para uma sub-exploração da mobilidade de código na construção de aplicações voltadas à Internet. Fato é, que poucos são os trabalhos que abordam tal questão, em virtude do enfoque dado às tecnologias envolvidas.

A construção de uma versão segura do ambiente μCODE, por meio da implementação do mecanismo de segurança descrito neste trabalho, possibilitou uma maior compreensão sobre as questões envolvidas na segurança das aplicações baseadas no uso de código móvel. Destaca-se a carência por publicações que relacionem as questões relativas à Segurança com o tema da Mobilidade de Código, estando o assunto restrito a um grupo relativamente pequeno de pesquisadores.

Vale ressaltar que, com relação ao ambiente μCODE, não foram encontrados trabalhos que co-relacionem os temas abordados neste trabalho, da mesma forma, não foram encontradas outras abordagens com o intuito de criar uma versão segura do ambiente em questão.

A implementação da versão segura para o ambiente μCODE permitiu a proteção dos servidores μServer dos chamados agentes maliosos, sendo a partir de então, oferecido aos usuários a possibilidade de definição, verificação e controle do comportamento de tais aplicações. Favoreceram a construção do mecanismo de segurança o uso da linguagem Java, pelo fato desta ser a linguagem nativa do ambiente. e a própria estrutura do

μCODE.

Por essa razão, optou-se pelo uso dos recursos de Assinatura de Código e Permissões. O uso do *Signed Group* possibilita verificar a origem de um determinado *Group*, enquanto que, por sua vez, as permissões associadas aos remetentes desses grupos assinados, disciplinam a forma como essas aplicações devem ser executadas. O mecanismo de segurança implementado foi completamente integrado ao ambiente original, ficando a cargo do desenvolvedor a escolha pelo uso da versão segura ou apenas dos recursos originalmente disponibilizados pelo ambiente.

Em seu atual estágio, a versão segura do ambiente μCODE, não encerra o assunto, havendo ainda muito por ser feito no sentido de expandir e melhorar as atuais características de segurança do ambiente. Entretanto, espera-se que este trabalho sirva de apoio a outras iniciativas que aprofundem a questão da Segurança em Código Móvel no Ambiente μCODE.

6.1 Trabalhos Futuros

Sugere-se como melhorias à atual versão segura do ambiente μCODE e como propostas de novos trabalhos nesta área:

- Realizar a validação do mecanismo de segurança implementado, através da construção de novas aplicações e realização de mais experimentos com o ambiente μCODE *Seguro*.
- Refinamento do mecanismo de segurança: através de melhorias do processo de tratamento das exceções e criação de guias de referências para programadores e desenvolvedores.
- Expansão do mecanismo de segurança: agregar ao μCODE *Seguro*, novas características que possibilitem aumentar as funcionalidades providas, como por exemplo, a implementação de mecanismos que protejam os agentes das chamadas *plataformas maliciosas*.
- Continuação dos trabalhos em nível de iniciação científica, visando o desenvolvimento de estudos e materiais de apoio a utilização da mobilidade de código..

Referências Bibliográficas

CARZANIGA, A.; PICCO, G. P.; VIGNA, G. Designing distributed applications with mobile code paradigms. *Proceedings of the 19th International Conference on Software Engineering - ICSE´97) - Boston, MA*, ACM Press, May 1997.

DEITEL, H. M.; DEITEL, P. J. *Java - Como Programar*. 3a. ed. Porto Alegre, RS: Bookman, 2001.

FRAGA, J. d. S.; WANGHAM, M. S. *(Mini-curso) Agentes Móveis X Segurança*. Universidade Federal de Santa Catarina - UFSC, 2001. Simpósio sobre Segurança em Informação - SSI 2001.

FUGGETTA, A.; PICCO, G. P.; VIGNA, G. Understanding code mobility. *IEEE Transactions on Software Engineering*, IEEE, v. 24, n. 5, May 1998.

GONG, L. *Inside Java 2 Plataform Security - Architecture, API Design and Implementation*. Palo Alto - CA - USA: Addison Wesley Longman Inc, 1999.

GOSLING, J.; MCGILTON, H. The java language environment. *Sun Microsystems*, Sun Microsystems Inc., May 1996.

GRITZALIS, S.; ILIADIS, J. Addressing security issues in programming languages for mobile code. *9th International Workshop on Database and Expert Systems Applications (DEXA'98)*, IEEE, Vienna, Austria August 1998.

MCGRAW, G.; FELTEN, E. W. Mobile code and security. *IEEE Internet Computing*, IEEE, November - December 1998.

MCGRAW, G.; FELTEN, E. W. *Securing Java - Getting Down to Business with Mobile Code*. 2nd edition. ed. [S.l.]: John Wiley Sons Inc, 1999.

NEWMAN, A. *Usando Java*. Rio de Janeiro, RJ: Campus, 1997.

PICCO, G. P. μCODE: A lightweight and flexible mobile code toolkit. In: *Mobile Agents - Proceedings of the 2nd International Workshop on Mobile Agents*. Stuttgart (Germany): K. Rothermel and F. Holh, 1998. (ISBN 3-540-64959-X, v. 1477), p. 160–171.

PICCO, G. P. Mobile agents: An introduction. *Journal of Microprocessors and MicroSystems*, Elsevier Science, 2001.

PICCO, G. P. *A Mobile Code Toolkit*. April 2001. Website (URL: http://mucode.sourgeforge.net).

RUBIN, A. D.; JR, D. R. G. Mobile code security. *IEEE Internet Computing*, IEEE, November - December 1998.

TANENBAUM, A. S.; STEEN, M. v. *Distributed Systems - Principles and Paradigms.* Upper Saddle River, New Jersey 07458: Prentice Hall, 2002.

ANEXO A – Programa: keyGen.java

```
import java.io.*;
import java.security.*;
class GenKey{
public static void main(String[] args){
if (args.length != 0) {
System.out.println("Uso: CPKey");
}
else try{
/* Geração do Par de Chaves */
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA", "SUN");
SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");
keyGen.initialize(1024, random);

KeyPair pair = keyGen.generateKeyPair();
PrivateKey priv = pair.getPrivate();
PublicKey pub = pair.getPublic();

/* Geração do arquivo que irá conter a Chave PRIVADA */
byte[] key = priv.getEncoded();
FileOutputStream keyfos = new FileOutputStream("privkey");
keyfos.write(key);

keyfos.close();

/* Geração do arquivo que irá conter a Chave PUBLICA */
byte[] key2 = priv.getEncoded();
```

```
FileOutputStream keyfos2 = new FileOutputStream("pubkey");
keyfos2.write(key2);

keyfos.close();

System.out.println("Arquivos Criados com SUCESSO!!!");
} catch (Exception e) {
System.err.println("Caught Exception " + e.toString());
e.printStackTrace();
}
}
};
```