Interface Básica para um Servidor Universal

Flávia Linhalis

Orientador: Prof. Dr. Dilvan de Abreu Moreira

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo – ICMC-USP, como parte dos requisitos para a obtenção do título de Mestre em Ciências – Área de Ciências de Computação e Matemática Computacional.

USP – São Carlos

Maio de 2000

“Se meditares sinceramente na provas que já venceste, nos problemas que já atravessaste, nas dores que já esqueceste e nos obstáculos que, muitas vezes, já contornaste, reconhecerás que o amparo de Deus esteve e está contigo em todos os momentos.”

Emmanuel

Agradecimentos

Agradeço primeiramente a Deus por ter me dado a oportunidade de estar aqui.

Agradeço à minha família, principalmente à minha mãe Iza e ao meu pai Luiz. Vocês sempre me apoiaram em minhas decisões e, apesar de saberem que ficaríamos ainda mais longe, não deixaram de me incentivar e me aconselhar nos momentos difíceis.

Agradeço à Maria Liz, Dani, Michele, Simone e Patrícia. Juntas formamos a animada República Me Ame, que me acolheu logo que cheguei em São Carlos. Em especial à Maria Liz, que preso como uma grande amiga.

Agradeço à Andrezza, Jorge e Mayb, os amigos que fiz logo que comecei o mestrado e que continuam em minha vida até hoje. À Majú e Selma que, juntamente com Andrezza, Jorge e eu, formaram a aconchegante Repúbllica Tcheca. À Mayb e Camillo, pela República Dois Mais Um e pelos famosos churrasquinhos.

Agradeço também ao Fredy, que foi quem me abriu as portas para este mestrado. Ao Dilvan que, mesmo a partir da metade do programa de mestrado, me orientou e me deu liberdade para trabalhar no projeto que mais me interessou.

Agradeço ao Grupo da Fraternidade Irmão Batuíra, que me acolheu e ajudou no meu aprendizado.

Agradeço ao Rudinei, o amor que encontrei no mestrado. Sua amizade, carinho e apoio foram muito importantes, principalmente na reta final do meu trabalho. Se você não for o último, sei que restará no mínimo uma boa amizade.

Agradeço à Selma e ao Waine por serem meus companheiros em meu primeiro emprego. A presença e amizade de vocês no meu dia a dia está sendo muito importante.

Agradeço àqueles que sorriram, que festejaram, que conversaram... que, de alguma forma, mesmo que por alguns instantes, fizeram parte da minha história durante o tempo em que estive neste mestrado.

Agradeço à CNPq pelo apoio financeiro.

Sumário

CAPÍTULO 1 - INTRODUÇÃO **1**

1.1 MOTIVAÇÃO 3

1.2 OBJETIVO 3

1.3 ORGANIZAÇÃO DA DISSERTAÇÃO 6

CAPÍTULO 2 - SERVIDORES UNIVERSAIS **9**

2.1 CONSIDERAÇÕES INICIAIS 9

2.2 CONSIDERAÇÕES SOBRE OS PARADIGMAS RELACIONAL E ORIENTADO A OBJETOS 9

2.3 OS PRINCIPAIS SERVIDORES UNIVERSAIS E SUAS POLÍTICAS DE SEGURANÇA 11

2.3.1 Segurança no Oracle8 Server *12*

2.3.2 Segurança no Informix Universal Server *12*

2.4 CONSIDERAÇÕES FINAIS 13

CAPÍTULO 3 - JAVA E OS AGENTES DE *SOFTWARE* **15**

3.1 CONSIDERAÇÕES INICIAIS 15

3.2 COMO DEFINIR UM AGENTE? 16

3.3 A LINGUAGEM DE PROGRAMAÇÃO JAVA 17

3.4 CARACTERÍSTICAS DE JAVA PARA O DESENVOLVIMENTO DE AGENTES 20

3.4.1 Java Beans *20*

3.4.1.1 Características dos Java *Beans* 21

3.4.1.2 Padrões de Projeto 21

3.4.2 Serialização *22*

3.4.3 Introspecção *23*

3.4.4 Class Loading *24*

3.5 CONSIDERAÇÕES FINAIS 24

CAPÍTULO 4 - MECANISMOS DE SEGURANÇA **27**

4.1 CONSIDERAÇÕES INICIAIS 27

4.2 AMEAÇAS E MÉTODOS DE ATAQUE 28

4.3 PROPRIEDADES DE SEGURANÇA 29
4.4 CRIPTOGRAFIA 30
4.4.1 Criptografia com Chave Secreta *32*
4.4.2 Criptografia Com Chave Pública *32*
4.5 ASSINATURA DIGITAL 33
4.5.1 Função Hash *34*
4.5.2 Certificação *35*
4.6 AUTORIZAÇÃO 37
4.7 CONSIDERAÇÕES FINAIS 37
CAPÍTULO 5 - A SEGURANÇA EM JAVA 39
5.1 CONSIDERAÇÕES INICIAIS 39
5.2 LINGUAGEM SEGURA 40
5.3 SEGURANÇA NA MÁQUINA VIRTUAL 41
5.4 SUPORTE A ASSINATURAS DIGITAIS 44
5.4.1 A API de Segurança de Java 2 .... *45*
5.4.2 As Ferramentas Java 2 para Assinatura Digital *46*
5.4.2.1 Os Arquivos JAR 46
5.4.2.2 A ferramenta *keytool* 46
5.4.2.3 A ferramenta *jarsigner* 48
5.5 DEFINIÇÃO DE POLÍTICAS DE SEGURANÇA 51
5.6 CONSIDERAÇÕES FINAIS 53
CAPÍTULO 6 - IMPLEMENTAÇÃO DA INTERFACE 57
6.1 CONSIDERAÇÕES INICIAIS 57
6.2 O AGENTE *GATEWAY* 58
6.3 OS AGENTES USUÁRIOS DA IBSU 60
6.4 O *SECURITYMANAGER* 61
6.5 O *POOL* DE AGENTES 62
6.6 A INTERFACE COM O BANCO DE DADOS 63
6.6.1 Manipulação de Grupos *64*
6.6.2 Manipulação de Roots *65*
6.6.3 Controle de Threads *66*
6.6.4 Implementação de um Banco de Dados *67*

6.7 O InfoAgent 69
6.8 Considerações Finais 71
CAPÍTULO 7 - CONCLUSÕES 73
7.1 Visão Geral 73
7.2 Resultados e Contribuições 74
7.3 Trabalhos futuros 74
REFERÊNCIAS BIBLIOGRÁFICAS 77

Lista de Figuras

FIGURA 1.1 - MODELO ARQUITETURAL CLIENTE/SERVIDOR 2

FIGURA 1.2 - VISÃO GERAL DA IBSU 5

FIGURA 2.1 - SGBDOR COMO SERVIDOR UNIVERSAL 11

FIGURA 3.1 - PROCESSO DE COMPILAÇÃO E EXECUÇÃO EM JAVA 18

FIGURA 4.1 - CARACTERÍSTICA PADRÃO DE UM MÉTODO DE CRIPTOGRAFIA 31

FIGURA 4.2 - CRIPTOGRAFIA COM CHAVE PARA CODIFICAÇÃO E DECODIFICAÇÃO 31

FIGURA 4.3 - GERAÇÃO DE ASSINATURA DIGITAL 34

FIGURA 4.4 – VERIFICAÇÃO DE ASSINATURA DIGITAL 35

FIGURA 5.1 - MODELO DE SEGURANÇA DO JDK 1.0 42

FIGURA 5.2 - MODELO DE SEGURANÇA DO JDK 1.1 43

FIGURA 5.3 - MODELO DE SEGURANÇA DO JDK 1.2 44

FIGURA 5.4 – ESTRUTURAÇÃO DO *KEYSTORE* 47

FIGURA 5.5 - ASSINATURA DE UM JAR 49

FIGURA 5.6 - VERIFICAÇÃO DA ASSINATURA DE UM JAR 50

FIGURA 5.7 – ESTRUTURAÇÃO DOS *POLICY FILES* 53

FIGURA 5.8 - PROCESSO PARA ASSINATURA DE APLICAÇÕES 54

FIGURA 5.9- PROCESSO DE AUTENTICAÇÃO E EXECUÇÃO DE APLICAÇÕES 55

FIGURA 6.1 - O *GATEWAY* 58

FIGURA 6.2 - O *POOL* DE AGENTES 62

FIGURA 6.3 - IMPLEMENTAÇÃO DO BANCO DE DADOS 69

FIGURA 6.4 - EXEMPLO DE PÁGINA ENVIADA PELO INFOAGENT 70

Resumo

Esse projeto implementa uma Interface Básica para um Servidor Universal (IBSU). A IBSU provê um ambiente para executar agentes de *software* e interfaces seguras entre estes e o banco de dados de um servidor universal. Os agentes têm acesso a *roots* (pontos de entrada para objetos) armazenados no banco de dados. A IBSU provê um ambiente aberto e seguro para a execução de agentes. Suas principais funções são receber os agentes, autenticá-los e prover acesso aos *roots* do banco de dados e aos recursos do sistema. Contudo, a IBSU não permitirá que um agente acesse um *root* ou recurso do sistema se este não tiver permissão para tal.

A IBSU é composta por quatro partes que garantem a abertura e segurança do ambiente de execução dos agentes: a Interface do Banco de Dados, o *Gateway*, o *Pool* de agentes e o Gerenciador de Segurança. A Interface do Banco de Dados define métodos que permitem aos agentes manipular grupos, *roots* e associar permissões de acesso entre *roots* e grupos no banco de dados. O *Gateway* recebe os agentes de *hosts* remotos e os autentica. Essa autenticação é feita verificando-se a assinatura digital, os certificados associados ao agente e a qual grupo ele pertence. Se o processo de autenticação tiver êxito, o agente pode se juntar ao *Pool*. O *Pool* de agentes executa os agentes e controla seu tempo total de vida. Esse tempo depende do grupo do qual esse agente faz parte. O Gerenciador de Segurança garante que os agentes que estão executando no *Pool* não conseguirão acesso a recursos que eles não tenham permissão para utilizar.

A IBSU e os agentes de *software* que executam no Pool são implementados em Java. Esses agentes têm, no servidor universal, o mesmo papel que as linguagens de consulta (como SQL) têm nos bancos de dados relacionais. Mas como eles têm a vantagem de possuir todo o poder de computação do ambiente Java (incluindo o maior poder de programação, abertura e segurança), eles podem realizam esse papel de forma muito mais eficiente.

Abstract

This project implements a Basic Interface for an Universal Server (BIUS). The BIUS is to be a place to host software agents and securely interface them with the database of an universal server. The agents will access the roots (entry points for objects) stored in the database. The BIUS provides an open and secure environment to software agents execution. Its main functions are receive the agents, authenticate them and provide access to database roots and systems resources. However, the IBSU will not allow that an agent accesses a root or a system resource if it does not have permission to do so.

The BIUS is composed by four parts that guarantee openness and security for the agents execution environment: the Database Interface, the Gateway, the Pool of Agents and the Security Manager. The Database Interface defines methods that allows the agents to handle groups, roots and associate access permissions between roots and groups in the database. The Gateway receives the agents from remote hosts and authenticates them. The authentication is done by verifying the digital signature, the certificate(s) associated with the agent and the group it belongs to. If the authentication process succeed, the agent can join the Pool. The Pool of Agents runs the agents and control their lifetime. The agent lifetime depends on witch group it belongs to. The Security Manager guarantees that running agents will not have access to Java platform resources they are no entitled to use.

The BIUS and the software agents that run in the Pool are implemented in Java. This agents have in the universal server the same role as query languages (such as SQL) have in relational database systems. But, as agents have the advantage of enjoying all the power provided by the Java environment (such as expression power, openness and security), they can fulfill this role much more efficiently.

Capítulo 1

Introdução

Um sistema distribuído [1, 2] pode ser caracterizado como sendo um conjunto de computadores independentes, interligados por uma rede de comunicação, que permite o compartilhamento dos diversos recursos do sistema, como *hardware*, *software* e dados. Além disso, o acesso aos recursos deve ser transparente ao usuário, ou seja, independente de sua localização, o usuário deve acessar os serviços da mesma forma, tendo a impressão de que todos os serviços estão em seu computador.

Atualmente, o modelo arquitetural dos sistemas distribuídos mais difundido é o cliente/servidor, pois ele apresenta uma arquitetura simplificada e flexível, que se adapta às necessidades da maioria dos usuários. A figura 1.1 ilustra o modelo cliente/servidor.

O modelo cliente/servidor é caracterizado pela presença de um ou mais servidores e várias estações de trabalho (clientes). O servidor é responsável pelo gerenciamento e disponibilização dos recursos da rede aos clientes que, por sua vez, realizam pedidos de serviço. É criada uma conexão com o servidor sempre que um cliente solicita um serviço. Além disso, o servidor é capaz de manter várias conexões com vários clientes, simultaneamente.

Cada conexão é feita especificando-se um endereço lógico denominado porta, que identifica o tipo de serviço a ser realizado. Após o recebimento do pedido, o servidor executa o serviço e envia o resultado de volta ao cliente.

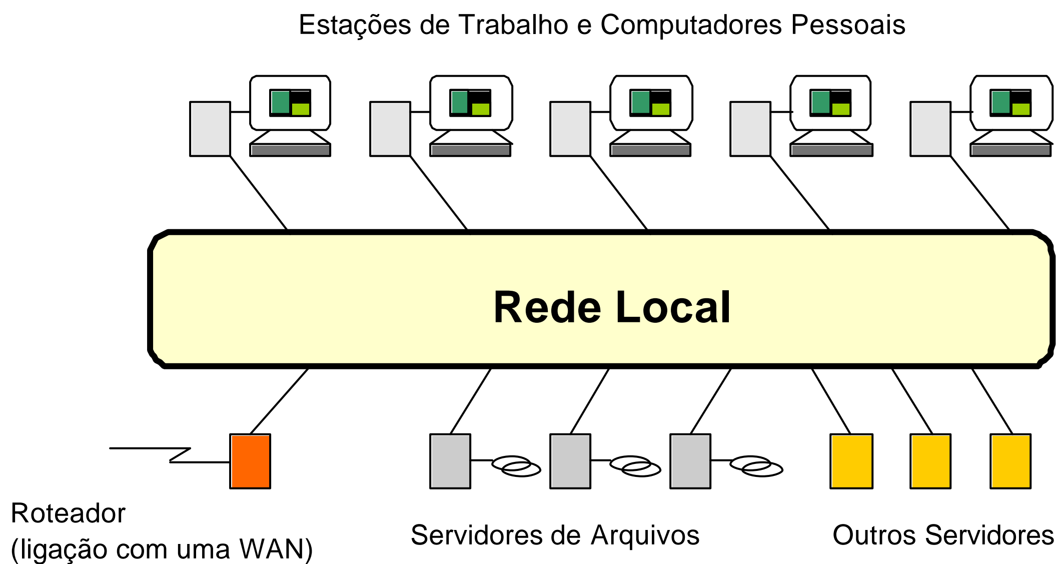

Figura 1.1 - Modelo arquitetural cliente/servidor

O Conceito de Servidor Universal

Com a difusão de sistemas distribuídos e do modelo cliente/servidor, existem atualmente muitos tipos de programas servidores de informação disponíveis, tais como servidores de HTML (*HyperText Markup Language*), de SQL (*Structured Query Language*), de FTP (*File Transfer Protocol*), etc. Todos eles possuem a mesma funcionalidade básica: fornecer informações a partir de requisições de clientes. Do ponto de vista do usuário, o que difere um tipo de servidor do outro são seus protocolos de comunicação, a maneira como a informação procurada deve ser especificada e como ela vem codificada.

A existência desses diversos tipos de servidores e, consequentemente, clientes obriga os *sites* que fornecem informações (bancos, universidades, *software houses*, etc.) a manterem vários tipos de servidores executando e, muitas vezes, a mesma informação em vários formatos diferentes. Seria então extremamente útil ter toda a informação em apenas um formato em um único banco de dados/servidor que pudesse fornecê-la em uma variedade de formatos e protocolos diferentes. Além disso, esse banco de dados/servidor poderia permitir que os usuários especificassem a informação procurada em uma variedade de formatos e maneiras. Esta é exatamente a idéia de um servidor universal.

Um servidor universal deve ser capaz de responder a diferentes tipos de requisições em diferentes portas utilizando o mesmo conjunto de informações. Para cada usuário, o fornecimento do serviço é transparente, isto é, parece que o servidor é dedicado a apenas aquele serviço em particular.

Para conseguir isso, todas as informações de interesse dos clientes devem ser mantidas em um único banco de dados, que pode ser lido e atualizado por qualquer serviço. Existirá, então, apenas uma fonte de dados a ser mantida e muitas maneiras de acessá-la e modificá-la.

1.1 Motivação

Como foi exposto, o desenvolvimento de servidores universais é de grande utilidade. Eles podem responder a diferentes tipos de requisições utilizando o mesmo conjunto de dados. Para que isso seja possível, os servidores universais possuem um único banco de dados, ou seja, as informações são armazenadas em um mesmo formato, mas podem ser fornecidas para os clientes em formatos e protocolos diferentes. O capítulo 2 descreve melhor alguns dos tipos de servidores universais existentes.

Como os dados armazenados em um servidor universal podem ter os mais variados objetivos, deve haver uma forma de controlar o acesso a eles. Assim, determinados dados serão acessados apenas por usuários/aplicações autorizados. É por isso que se torna necessária a presença de uma interface entre o banco de dados do servidor universal e os usuários/aplicações que terão acesso aos dados armazenados.

1.2 Objetivo

O objetivo do projeto descrito nesta dissertação é o desenvolvimento de uma **Interface Básica para Servidor Universal** (IBSU). Este projeto enfoca a segurança para acesso aos dados de um servidor universal em nível de autenticação e autorização. Para isso, a IBSU deve estar posicionada entre o banco de dados, que faz parte do servidor universal, e as aplicações que desejarem acessar os objetos que ele armazena. As aplicações são desenvolvidas como agentes de *software* (ver capítulo 3) que, de forma autônoma, irão acessar o banco de dados através dos métodos definidos em uma interface com o banco de

dados. Considera-se o banco de dados como parte de um servidor universal porque os agentes que irão acessá-lo podem ser de qualquer tipo e, por isso, podem acessar os mais diversos tipos de objetos e servi-los a seus clientes das mais diferentes formas. A função da IBSU é, basicamente, receber os agentes, controlar o acesso dos agentes aos dados (*roots)* armazenados no banco de dados do servidor universal e aos recursos do sistema.

A palavra *root*, nesta dissertação, refere-se a um objeto armazenado no banco de dados. A escolha da palavra *root*, traduzida como origem, se deu porque um *root* é simplesmente o ponto de entrada (ou origem) de um objeto. Esse objeto pode referenciar outros objetos a partir de sua origem. O *root* é a porta de acesso ao objeto como um todo. Por exemplo, se uma árvore é um objeto, seu nodo principal é o *root*, pois a partir dele pode-se acessar todos os outros nodos. Esta dissertação considera todos os objetos armazenados no banco de dados como estando em *roots.*

A autenticação feita pela IBSU diz respeito à validação da identidade do possuidor de um agente. A autorização define que privilégios um agente terá para realizar acesso aos *roots* e aos recursos do sistema. A figura 1.2 dá uma visão geral da IBSU desenvolvida neste projeto.

De acordo com a figura 1.2, a IBSU é composta de quatro componentes: a Interface com o Banco de Dados, o *Gateway*, o *Pool* de Agentes e o *Security Manager*.

Para acessar os *roots*, um agente deve utilizar-se da Interface com o Banco de Dados. Essa interface possui a definição de métodos que permitem ao agente criar grupos, criar *roots* e associar permissões de acesso entre os *roots* e os grupos. Assim, um agente só poderá acessar os *roots* que a ele forem permitidos, o que dependerá das permissões concedidas ao grupo que ele pertence.

Se os métodos da Interface com o Banco de Dados fossem acessados exclusivamente por usuários locais, bastaria o controle de acesso para garantir a segurança dos *roots.* Mas, como serão aplicações desenvolvidas como agentes de *software* móveis a entrar em contato com os métodos, torna-se necessário autenticar os agentes, pois não é nada seguro para o sistema permitir que agentes desconhecidos entrem em execução. Essa autenticação é feita pelo agente *Gateway*. Ele recebe agentes vindos de *hosts* desconhecidos, checa a assinatura digital e os certificados (ver capítulo 4) daquele agente. A checagem da assinatura irá garantir que os arquivos que compõem a aplicação agente não foram alterados depois que a assinatura

foi gerada. Irá garantir também, que a assinatura foi realmente gerada (codificação) com a chave privada correspondente à chave pública que está sendo utilizada para a verificação (decodificação). A utilização de uma chave privada para codificação dos arquivos do agente é uma evidência de identidade, pois a chave é conhecida apenas pelo seu possuidor. O *Gateway* também verifica o(s) certificado(s) associado(s) à chave pública. Caso a verificação seja feita com sucesso, significa que o dono daquele agente é confiável.

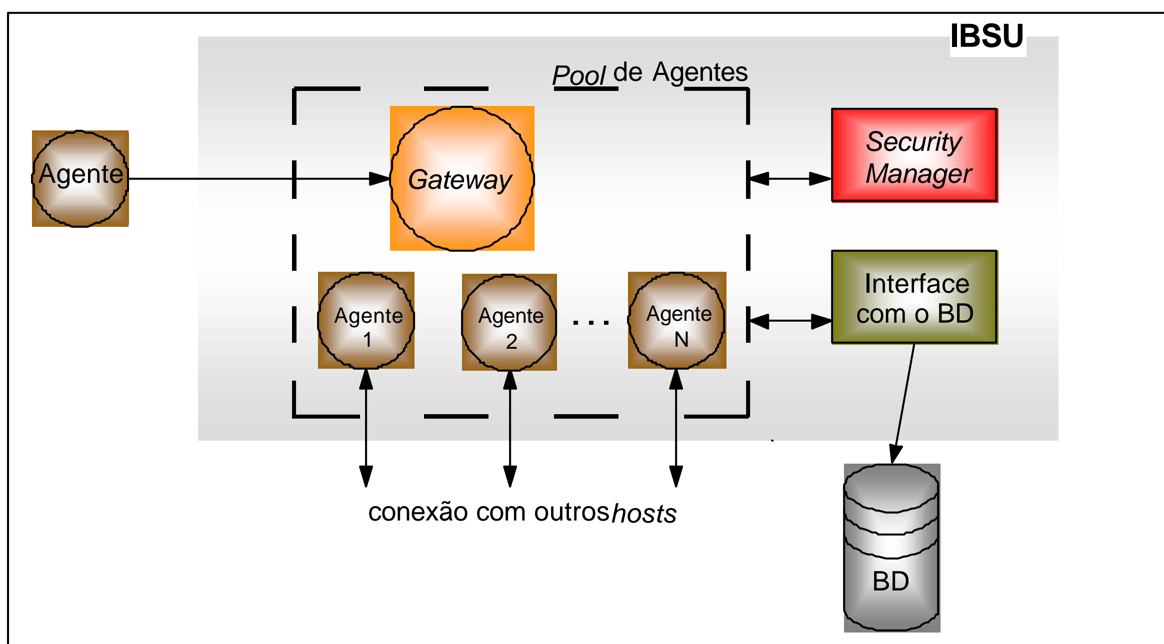

Figura 1.2 - Visão geral da IBSU

Caso um agente esteja corretamente assinado e possua pelo menos um certificado confiável, ele poderá entrar em execução no *Pool* de Agentes. A função do *Pool* é colocar os agentes em execução e controlar o tempo de vida deles. O tempo de execução de cada agente depende do grupo ao qual ele pertence.

Como o *Gateway* é um agente, ele também deve ser controlado pelo *Pool*. O *Gateway* é independente dos demais agentes que estiverem em execução. Por isso, se ele for retirado do *Pool*, os outros agentes continuarão em execução. Isso implicaria apenas na não entrada de novos agentes no *Pool*.

A IBSU e os agentes que ganharão acesso ao *Pool* são desenvolvidos utilizando-se a linguagem de programação Java. Por isso, os agentes podem utilizar-se de todos os recursos

que a plataforma Java tem a oferecer. Eles poderiam, por exemplo, ler e escrever no sistema de arquivos, estabelecer conexões com *hosts* remotos, modificar propriedades do sistema, etc. Mas, por questão de segurança, nem todos os grupos devem ter acesso a todos os recursos do sistema. Por isso, a classe *SecurityManager* de Java é instalada. O *SecurityManager* deve controlar o acesso dos agentes aos recursos do sistema de acordo com as permissões que estiverem associadas ao grupo do agente.

Como a IBSU e os agentes são desenvolvidos em Java, ela se torna, naturalmente, a linguagem de consulta ideal para o banco de dados. Vale a pena observar o poder da substituição das linguagens de consulta pelo uso de agentes. Com isso, o usuário não está limitado a nenhuma linguagem declarativa de consulta, o que pode aumentar grandemente a capacidade de uma aplicação. Ela pode criar sua própria linguagem para consulta e até mesmo mudar como o servidor é visto pela rede (de um servidor universal em uma porta N, para um servidor de HTTP na porta 80, por exemplo).

A autenticação realizada pelo *Gateway*, o controle de acesso aos *roots*, proporcionado pela Inteface com o Banco de Dados, e o controle de acesso aos recursos do sistema, feito pelo *SecurityManager*, possibilitam que o *Pool* seja um ambiente de execução de agentes aberto e ao mesmo tempo seguro. Os agentes poderão se conectar a recursos externos, carregar objetos no banco de dados e realizar uma série de outras operações de forma segura, ou seja, de acordo com os privilégios concedidos ao seu grupo.

1.3 Organização da Dissertação

O capítulo 2 apresenta algumas considerações a respeito dos servidores universais e as políticas de segurança adotadas pelos principais servidores universais existentes.

No capítulo 3 é apresentada a linguagem de programação Java, que é utilizada no desenvolvimento deste projeto, a tecnologia de agentes de *software* e as vantagens que Java pode oferecer para o desenvolvimento de agentes.

O capítulo 4 apresenta um estudo teórico sobre os mecanismos de segurança existentes, o que engloba criptografia, autenticação e controle de acesso.

O capítulo 5 trata de segurança em Java. São apresentadas as facilidades que Java fornece aos desenvolvedores que desejam implementar aplicações seguras. Nesse capítulo é descrito como Java implementa os mecanismos de segurança descritos no capítulo 4.

No capítulo 6 é descrito como este projeto foi implementado. Cada módulo da IBSU é detalhado, mostrando como os recursos de Java foram utilizados. Descreve-se também qual deve ser o perfil dos agentes que entrarão em execução no *Pool* e, para finalizar, apresenta-se o InfoAgent, um agente que demonstra a funcionalidade da IBSU.

O Capítulo 7 apresenta as conclusões deste trabalho e propõe trabalhos futuros.

Capítulo 2

Servidores Universais

2.1 Considerações Iniciais

Como exposto no capítulo anterior, um servidor universal deve ser capaz de armazenar os mais diversos tipos de dados na forma de objetos. Os sistemas atuais tornaram-se mais complexos e requerem variados tipos de objetos, tais como imagens, vídeo, ou mesmo páginas da *Web* (as quais podem ser vistas como objetos complexos contendo objetos mais simples inter-relacionados).

Este capítulo tem por objetivo apresentar os Sistemas Gerenciadores de Banco de Dados (SGBD) que suportam o armazenamento dos objetos complexos de um servidor universal. São apresentadas também as políticas de segurança adotadas pelos principais servidores universais existentes no mercado: o *Oracle8 Server* e o *Informix Universal Server*.

2.2 Considerações sobre os Paradigmas Relacional e Orientado a Objetos

A indústria de bancos de dados oferece duas maneiras para a manipulação de objetos complexos, as quais podem ser utilizadas para armazenar os dados de um servidor universal: os Sistemas Gerenciadores de Banco de Dados Orientados a Objeto (SGBDOO) e os Sistemas

Gerenciadores de Banco de Dados Objeto-Relacionais (SGBDOR), que são uma extensão dos Sistemas Gerenciadores de Banco de Dados Relacionais (SGBDR). Os métodos definidos na Interface com o Banco de Dados da IBSU poderiam ser implementados para acessar bancos de dados pertencente a um SGBDOO ou um SGBDOR.

Os SGBDOO estenderam as linguagens de programação orientadas a objeto para prover características necessárias a um banco de dados. Os SGBDOO têm se tornado uma boa opção para manipular objetos de dados complexos, pois seu paradigma se adequa naturalmente às necessidades dos objetos complexos. As características que tornam vantajosa a abordagem orientada a objetos são as seguintes:

- **O Paradigma de Orientação a Objetos**: os SGBDOOs geralmente fornecem uma linguagem para consulta integrada com a linguagem de programação, e assim podem compartilhar o mesmo sistema de tipos para os dados. Essa extensão se aproveita da força do paradigma da orientação a objetos, já que as classes dos objetos utilizados pela linguagem de programação são as mesmas classes utilizadas pelo SGBDOO. Devido a consistência desse modelo, não há necessidade de transformar os objetos da aplicação para que eles possam ser armazenados no banco de dados, como seria preciso em um SGBDR [3].

- **Produtividade**: o tempo de desenvolvimento em um SGBDOO é menor, pois não há necessidade de escrever o código requerido para enquadrar objetos em tabelas [4].

- **Performance**: a performance de um SGDBOO é melhor quando os objetos são complexos. A incapacidade de representação desses objetos em um SGBDR pode levar à fragmentação de um objeto em várias tabelas. Para recuperar informação sobre esse objeto complexo por inteiro, são necessárias várias operações de *join* nas tabelas, o que degrada a performance em um SGBDR.

Os SGBDR têm sido adaptados para acomodar os novos requisitos dos objetos. Esses sistemas, chamados de SGBDOR, estenderam os sistemas relacionais para prover mecanismos que possibilitem às linguagens de consulta dos SGBDR manipularem objetos. Para Rogers [3], os banco de dados objeto-relacionais são apenas bancos de dados relacionais "disfarçados", capazes de armazenas objetos grandes, e tudo isso não passa de uma estratégia de *marketing*.

2.3 Os Principais Servidores Universais e suas Políticas de Segurança

Como este projeto enfoca a segurança para acesso aos dados de um servidor universal, nesta seção são brevemente apresentados dois dos principais servidores universais existentes no mercado e como eles implementam suas políticas de segurança.

Como visto na seção anterior, os SGBDOO seriam os mais adequados para o armazenamento dos dados de um servidor universal. Mas, estratégia de *marketing* ou não, o mercado mostra que os usuários dos SGBDOO ainda são pequenos nichos e que os SGBDOR continuam com a preferência da maioria dos usuários [4].

Os servidores universais que ganharam mais espaço são o *Oracle8 Server* e o *Informix Universal Server* [6]. De acordo com o Grupo Aberdeem [5], um servidor universal é um SGBDR que oferece aos usuários a habilidade para acessar tipos de dados complexos, ou seja, qualquer SGBDOR pode ser um servidor universal. A figura 2.1 ilustra um SGBDOR como servidor universal.

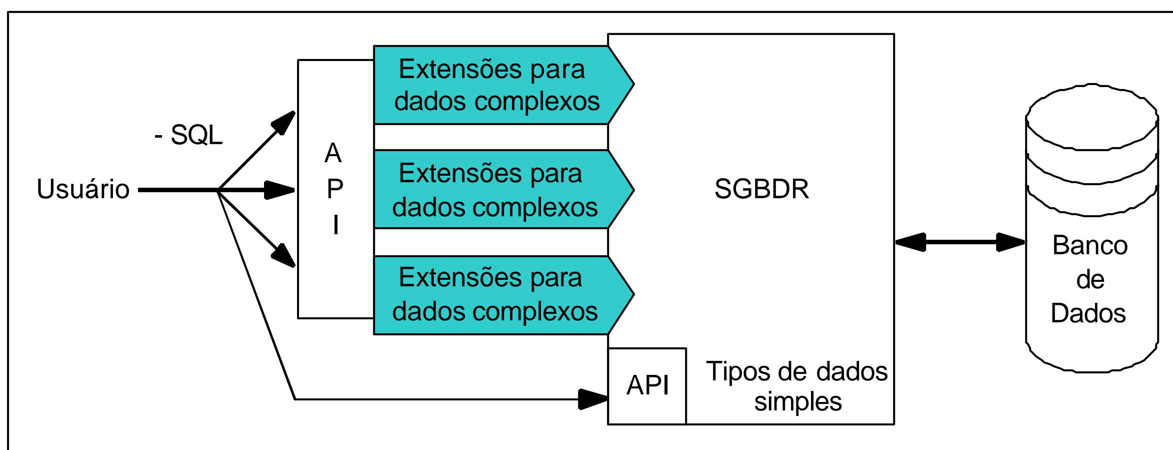

Figura 2.1 - SGBDOR como servidor universal

Como pode ser observado, os servidores universais implementados a partir dos SGBDR acrescentam suporte a objetos complexos, o que os torna aptos a armazenar qualquer tipo de objeto definido pelo usuário. Tanto o *Oracle8 Server* quanto o *Informix Universal*

Server suportam funções definidas em linguagens de programação (Java, PowerBuilder, Visual Basic, C/C++), mas continuam a utilizar SQL3 como linguagem de consulta.

2.3.1 Segurança no *Oracle8 Server*

O *Oracle8 Server* possui o *Oracle Security Sever* [7], que é um produto para tratar exclusivamente da segurança em um ambiente *Oracle*. O *Oracle Security Server* suporta autenticação e autorização. A autenticação garante que a identidade da entidade que deseja acessar um ou mais servidores *Oracle* é verdadeira. A autorização assegura que uma dada entidade só pode operar de acordo com os privilégios definidos pelo administrador. Para isso, o *Oracle Security Server*, em sua versão 2.0.3, implementa criptografia e autenticação (ver capítulo 4) da seguinte forma:

- Fornece um ambiente para autorização e autenticação baseada em criptografia de chave pública. Suporta a versão 1 dos certificados do tipo X.509 [8], que é um método padrão para autenticação.
- Possui o *Oracle Security Server Manager*, uma ferramenta de gerenciamento que o administrador utiliza para configurar o ambiente.
- Possui o *Oracle Cryptographic Toolkit* para o programador. Esse *toolkit* contém um conjunto de APIs (*Application Programming Interface*) que permitem que os programas de aplicação acessem funções de criptografia, tais como para geração e verificação de assinaturas digitais. Essas APIs estão disponíveis via *Oracle Call Interface* (OCI) e PL/SQL. Elas podem prover segurança para uma grande variedade de aplicações, tais como correio eletrônico e comércio eletrônico.

2.3.2 Segurança no *Informix Universal Server*

No *Informix Universal Server* a autenticação dos usuários é feita através de um *login* válido no arquivo de *passwords* do sistema operacional. O usuário deve ser membro de um grupo válido do sistema operacional e também de um grupo que tenha permissão para acessar o banco de dados [9]. O controle de acesso pode ser feito em vários níveis, o que inclui bancos de dados e tabelas específicas.

Além da autenticação via *password* e do controle de acesso, existe o *Informix-Universal Server Secure-Auditing Facility* ou simplesmente *auditing*, que é uma forma de controlar as ações dos usuários.

O *auditing* cria um registro das atividades executadas por determinados usuários. As atividades e os usuários são definidos previamente. Os registros de *auditing* podem ser utilizados para os seguintes propósitos:

- detectar ações não convencionais ou suspeitas e identificar qual usuário executou aquelas ações;
- detectar tentativas de acesso não autorizadas;
- avaliar potenciais danos na segurança;
- fornecer evidências em investigações, se necessário;
- fornecer uma maneira pacífica de impedir ações não desejadas, uma vez que os usuários sabem que suas ações podem estar submetidas ao *auditing*.

Mesmo podendo contar com o *auditing*, a autenticação via *password* do *Informix Universal Server* não garante a autenticidade do usuário.

2.4 Considerações Finais

De acordo com o exposto neste capítulo, pode-se fazer um paralelo dos servidores universais analisados com a abordagem adotada pela IBSU, levando-se em consideração a segurança e a linguagem de consulta.

Com relação a segurança, o *Oracle8 Server* promove autenticação e autorização, assim como a IBSU. O mesmo não pode ser afirmado a respeito do *Informix Universal Server*, onde não há garantia sobre a autenticidade dos usuários.

A linguagem de consulta utilizada pelos dois servidores universais líderes de mercado é SQL3. A IBSU recebe e executa agentes desenvolvidos em Java, o que faz de Java a linguagem de consulta. Sabe-se que Java é muito mais completa e flexível quando comparada a uma linguagem utilizada especificadamente para consulta, como é o caso de SQL3. Tendo-

se Java como linguagem de consulta pode-se aumentar grandemente a capacidade de uma aplicação. Além disso, a abordagem adotada torna o sistema aberto, pois um agente pode utilizar-se dos recursos da linguagem Java para acesso a recursos remotos, sob a supervisão do *Security Manager*.

Capítulo 3

Java e os Agentes de *Software*

3.1 Considerações Iniciais

Os agentes de *software* [10] foram inventados para facilitar a criação de *softwares* capazes de interoperar, ou seja, trocar informações e serviços com outros programas e, dessa forma, resolver problemas complexos.

A metáfora utilizada pelos agentes de *software* é a de um assistente que colabora com o usuário e/ou outros agentes. O conjunto de tarefas ou aplicações nas quais um agente pode auxiliar é praticamente ilimitado: filtragem de informações, obtenção de informações em bancos de dados, gerenciamento de *e-mail*, escalonamento de reuniões, seleção de livros, filmes, música, etc[11]. Neste capítulo são apresentadas as características que um programa deve apresentar para ser considerado um agente. É apresentada também a linguagem de programação Java e as facilidades que uma linguagem orientada a objetos pode proporcionar.

Além disso, são descritos alguns recursos da linguagem Java que a tornam apropriada para o desenvolvimento de agentes de *software* e, consequentemente, adequada como linguagem de consulta utilizada por agentes. Ou seja, ao invés de utilizar uma linguagem pré estabelecida para consulta, como SQL, um agente implementado em Java pode utilizar-se da própria linguagem Java para obter e modificar objetos armazenados.

3.2 Como Definir um Agente?

Pesquisadores que trabalham com agentes de *software* têm oferecido uma grande variedade de definições para o termo, cada um procurando explicar a sua própria utilização da palavra "agente". Por isso, definições para o mesmo termo podem ser diferentes, o que, muitas vezes, gera confusão.

De acordo com Maes [12]:

"Agentes autônomos são sistemas computacionais que habitam algum ambiente complexo dinâmico, sentem e agem de maneira autônoma nesse ambiente e, assim, atingem objetivos ou cumprem tarefas para as quais foram designados."

Para Smith et. al. [13] um agente é

"...uma entidade de *software* persistente dedicada a um propósito específico. Persistência distingue agentes de subrotinas; agentes têm suas próprias idéias sobre como realizar tarefas, sua própria agenda. Propósito específico os distingue da maioria das demais aplicações; os agentes são tipicamente menores."

Segundo Jennings e Wooldridge apud Franklin e Graesser [14], um agente é

"...um *hardware* ou (mais usualmente) um sistema computacional baseado em *software* que têm como características autonomia, capacidade social, reatividade e pro-atividade."

O termo agente não tem uma definição consensual, talvez por cobrir diversas áreas de investigação e desenvolvimento. As várias definições de agentes fornecem uma lista de atributos para eles, o que não significa que todos os agentes tenham que, necessariamente, conter todos os atributos. Os atributos de um agente dependem do tipo de aplicação que está sendo desenvolvida[15]. Segundo Franklin e Graesser [14], as caraterísticas encontradas na maioria dos agentes são :

- **Autonomia**: um agente será tão mais autônomo, quanto mais controle tiver sobre as suas ações. Um agente pode ser considerado autônomo em relação ao ambiente ou em relação a outros agentes.

- **Pró-atividade**: um agente pró-ativo toma a iniciativa para atingir os seus objetivos, não se limitando a responder a estímulos do ambiente.

- **Reatividade**: um agente tem capacidade de reagir às mudanças que sente no ambiente (estímulos).

- **Continuidade Temporal**: um agente está continuamente ativo. Nota-se que grande parte dos *softwares* existentes não tem essa característica, já que executam uma ou mais tarefas e terminam.

- **Capacidade Social**: se um agente tem capacidade social, então ele se comunica com outros agentes, o que poderá incluir humanos. Dessa comunicação poderá resultar uma cooperação. Para o caso específico da comunicação entre o agente e o usuário humano deverá ocorrer uma cooperação na construção do "contato" sobre o que o agente deverá fazer e não uma simples ordem.

- **Capacidade de adaptação**: um agente com capacidade de adaptação é capaz de alterar seu comportamento com base na experiência. Esse tipo de agente é o chamado "agente inteligente". Assim, diz-se também que esse agente tem capacidade de aprendizagem. A adaptação pode ser relativa ao ambiente ou no sentido de melhorar a sua interação com outros agentes.

- **Mobilidade**: corresponde à capacidade do agente se mover dentro do ambiente. Um agente móvel é capaz de se transportar de uma máquina para outra durante a sua execução.

- **Flexibilidade**: um agente com flexibilidade é aquele que não executa ações pré definidas em roteiros. Ou seja, possui a capacidade de escolher dinamicamente as ações e a seqüência de ações das mesmas, em resposta a um estado do ambiente.

- **Caráter**: possui personalidade e estado emocional.

Para Franklin e Graesser [14], autonomia, pró-atividade, reatividade e continuidade temporal são características essenciais em um agente.

3.3 A Linguagem de Programação Java

A linguagem de programação Java [16, 17] foi concebida em 1990 por James Gosling da *Sun Microsystems*. O objetivo inicial era criar um *software* para controle de aparelhos eletrodomésticos. Depois de algumas experiências frustrantes com C++, surgiu Java - a linguagem, bibliotecas e máquina virtual. Java ganhou popularidade rapidamente e, em 1995, a *Netscape* e a *Microsoft* a incorporaram em seus *browsers*, o que impulsionou grandemente a

difusão da linguagem. O sucesso de Java não se deu por ela ser poderosa ou sofisticada, mas porque o conjunto de suas características a torna simples, concisa e portátil.

A portabilidade de Java torna possível que um programa escrito nessa linguagem seja executado em qualquer plataforma sem a necessidade de alteração no código. Tal portabilidade é possível devido ao processo de compilação e execução em Java. O compilador Java traduz programas fontes em um código intermediário e independente de plataforma chamado *byte code*, que é interpretado pela *Java Virtual Machine* (JVM). A figura 3.1 ilustra o processo de compilação e execução em Java.

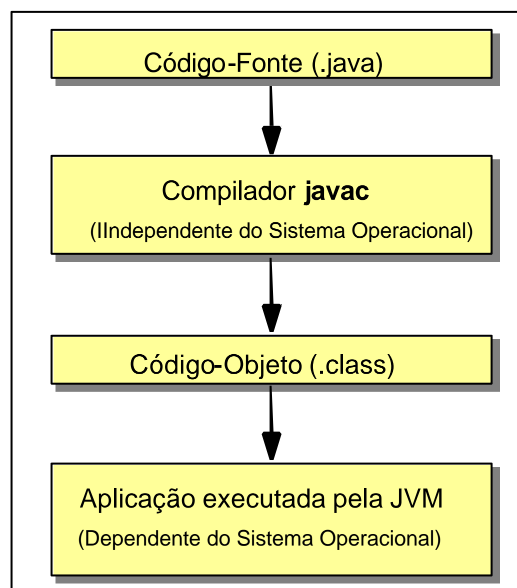

Figura 3.1 - Processo de compilação e execução em Java

Java é uma linguagem robusta e de propósito geral, o que inclui suporte à programação distribuída em alto nível e *multithreading*. A facilidade de programação combinada com a segurança, ajuda no rápido desenvolvimento de aplicações. Erros de programação comuns ocasionados por falha de alocação de memória, ponteiros e incompatibilidade de objetos não ocorrem em Java. Além disso, os mecanismos para manipulação de exceções [18] fornecem um modelo para lidar com erros de execução rápida e eficientemente. A segurança em Java será melhor abordada no capítulo 5.

Java e os Recursos da Orientação a Objetos

Java é orientada a objetos, o que permite extensibilidade e encapsulamento de dados. A extensibilidade [19] envolve a criação de uma nova solução, a partir de uma já existente, apenas programando as diferenças. Assim, um sistema pode crescer de forma econômica e facilitada. A extensibilidade só é possível através da reusabilidade de código. A orientação a objetos fornece três mecanismos para reusabilidade e extensibilidade: classes, herança e polimorfismo. Java fornece suporte aos três.

Uma classe é um espaço para a definição de métodos e variáveis para um tipo particular de objeto. A herança (ou *subclassing*) ocorre quando uma classe utiliza os métodos e variáveis definidas em outra classe. A classe que herdou é chamada subclasse e a classe onde os métodos e variáveis foram definidos é chamada de superclasse.

O polimorfismo está diretamente relacionado com a ligação dinâmica (*dynamic binding*). Polimorfismo é a habilidade de servir a um mesmo propósito com mais de uma maneira. Assim, o mesmo nome pode ser dado a métodos diferentes. Isso só é possível devido à ligação dinâmica, que é um mecanismo para determinar o tipo de um objeto em tempo de execução para, então, chamar o método apropriado. Não é necessário nenhum procedimento especial para a declaração de métodos polimórficos em Java. O *Java Runtime Environment* (JRE) cuida da ligação dinâmica em tempo de execução.

Segundo Rumbaugh et. al. [20], encapsulamento é

"...uma técnica de modelagem e implementação que separa os aspectos externos de um objeto dos detalhes internos de implementação."

O encapsulamento não é uma vantagem apenas das linguagens orientadas a objeto, mas essas já são beneficiadas naturalmente devido à estruturação em classes. A declaração de variáveis e métodos dentro de uma classe produz um tipo de dado que une estado e comportamento, mas é necessário que existam regras de controle de acesso que estabeleçam limites dentro da classe. O encapsulamento se dá através da combinação da classe com as regras de controle de acesso estabelecidas para suas variáveis e métodos. Assim, certos detalhes de implementação podem ser escondidos das aplicações clientes.

3.4 Características de Java para o Desenvolvimento de Agentes

A portabilidade combinada com a facilidade de programação tornam Java a linguagem escolhida por vários sistemas de agentes, em especial os agentes móveis, os quais devem ser capazes de migrar de uma plataforma a outra. Grandes sistemas de agentes móveis como *Concordia*, *Odyssey* e *Voyager* são desenvolvidos em Java [21]. Além da portabilidade, Java possui várias outras características adequadas para a implementação de agentes, entre elas: Java *Beans*, serialização, introspeção, *class loading* e segurança. Nas seções seguintes são descritas cada uma delas, com exceção da segurança, que possui o capítulo 5 inteiramente dedicado para si, já que este projeto focaliza a segurança em especial.

3.4.1 Java *Beans*

Como já foi exposto, um dos objetivos dos agentes de *software* é facilitar a criação de programas capazes de interagir e, assim, resolver problemas complexos. Um passo nessa direção é o desenvolvimento de arquiteturas de componentes reutilizáveis, o que permite que grandes sistemas sejam projetados combinando-se componentes pequenos e simples de origens possivelmente diferentes.

A linguagem de programação Java tem o potencial para eliminar *softwares* “gordos”, os quais possuem centenas de linha de código em um mesmo arquivo, o que na maioria das vezes gera redundância de código. Esses tipos de *softwares* são largamente encontrados nos dias de hoje. A natureza orientada a objetos de Java pode facilitar a criação de componentes de *software* “enxutos”, pois um componente pode ser um bloco de código embutido em uma classe.

Com os componentes reutilizáveis, veio à tona a programação visual, através da qual pode-se arrastar “pedaços de código” (componentes visuais, como botões no *Borland Delphi*) para dentro de um *form* e fazer com que eles trabalhem juntos. Java levou a criação de componentes visuais a um estado bastante avançado através de Java *Beans* [17, 22], pois um *bean* é simplesmente uma classe. Não é necessário escrever nenhum código extra ou usar extensões especiais da linguagem para fazer de uma classe um *bean*. A única necessidade é

modificar a maneira de dar nome aos métodos, seguindo alguns padrões de projeto (seção 3.4.1.2), e implementar a interface *Serializable*.

3.4.1.1 Características dos Java *Beans*

Os *beans* variam na funcionalidade que suportam, mas as características típicas que distinguem um *bean* de outro são: propriedades, métodos, personalização, eventos, persistência e introspeção.

As propriedades (ou atributos) são variáveis que podem afetar a aparência e o comportamento do *bean*. Elas podem ser lidas ou modificadas através da chamada de métodos apropriados no *bean*. As propriedades tipicamente são persistentes e podem também ser personalizadas em relação a aparência e ao comportamento. Um *bean*, como qualquer outra classe, possui um conjunto de métodos, os quais podem ser chamados por outros componentes.

Um *bean* também pode disparar eventos. Assim, um componente pode avisar a outros componentes que algo interessante aconteceu. Eventos são mecanismos para propagar notificações de mudanças de estado entre o objeto origem e um ou mais objetos destino (*listeners*). Assim, eventos fornecem um mecanismo conveniente para permitir que componentes sejam ligados.

As características de persistência e introspecção serão melhor abordadas nas seções 3.4.2 e 3.4.3, respectivamente.

3.4.1.2 Padrões de Projeto

O termo padrão de projeto significa que existem nomes e tipos convencionais para métodos e/ou interfaces, que são utilizados por motivos de padronização. Os padrões de projeto possuem duas utilidades. Primeiro, eles são boas sugestões de documentação para programadores. Identificando métodos com padrões de projeto apropriados, o programador pode assimilar e utilizar novas classes mais rapidamente. Segundo, pode-se escrever ferramentas e bibliotecas que reconheçam padrões de projeto e os utilizem para analisar e compreender componentes. Para Java *Beans*, utiliza-se identificação automática de padrões de projeto como um caminho para as ferramentas identificarem propriedades, eventos e métodos exportados.

Em um *bean*, para uma variável ou propriedade chamada `xxx`, tipicamente cria-se dois métodos: `getXxx()` e `setXxx()`. Sendo que o primeiro obtém o valor da variável `xxx` e o segundo modifica o valor de `xxx`. No caso dos eventos, deve-se criar um objeto "*listener*" e registrá-lo com o componente que está disparando o evento. Esse registro é feito através da chamada ao método `addxxxListener()` no componente de onde o evento é disparado, onde `xxx` representa o tipo do evento [17].

É importante observar que a abordagem utilizada por Java *Beans* torna possível aos *beans* auxiliar a linguagen Java no papel de linguagem de consulta a banco de dados. Através de seus métodos *get* e *set*, a informação armazenada em um banco de dados pode ser obtida e modificada. Nesse projeto todos os objetos armazenados no banco de dados são considerados *beans*.

3.4.2 Serialização

A capacidade de armazenar e recuperar objetos Java é chamada Serialização [17, 23]. Para isso, Java permite que os objetos sejam salvos na forma de um *stream*, ou seja, uma seqüência de *bytes* que descreve totalmente o objeto. É muito simples promover a serialização de uma classe, basta que ela implemente a interface `Serializable`.

Os *beans*, em especial, tem a necessidade de suportar armazenamento, pois quando um *bean* é utilizado, as informações a respeito do seu estado são geralmente configuradas em tempo de projeto. Essas informações necessitam ser armazenadas e posteriormente recuperadas, para dar a mesma aparência e comportamento ao *bean*, quando o programa for iniciado. Normalmente, um *bean* irá armazenar todas as suas propriedades. Ele pode também armazenar estados internos adicionais que não são diretamente acessíveis via propriedades, o que pode incluir personalização. É por isso que, para ser considerada um *bean*, além de seguir os padrões de projetos, uma classe deve ser serializável.

Entre as vantagens em na serialização de objetos Java pode-se citar:

- A portabilidade necessária para objetos remotos. Ou seja, pode-se criar um objeto em uma máquina Windows, serializá-lo e enviá-lo através da rede até uma máquina UNIX, onde ele será corretamente reconstruído. Assim, não é necessário se preocupar com as diferentes representações de dados, com a ordem dos *bytes* e

outros tipos de detalhes. Essa portabilidade é muito importante para os agentes móveis, pois eles podem migrar sem ter o conhecimento das características dos computadores onde irão entrar em execução.

- Suporte a persistência. Uma classe só poderá ser armazenada de forma persistente se ela for serializável. Nesse caso, uma classe serializável pode, até mesmo, fazer o papel de banco de dados para uma aplicação.

3.4.3 Introspecção

Em programação baseada em componentes é comum utilizar-se uma ferramenta visual para a construção da aplicação. Com essa maneira visual de criar programas (movendo-se ícones que representam componentes) torna-se necessária a configuração das propriedades dos objetos em tempo de programação. Isso requer que um componente possa expor algumas de suas propriedades para que sejam lidas e selecionadas.

No passado, um componente de *software* deveria publicar a definição de sua API em um arquivo separado ou utilizar alguma outra técnica para informar as outras aplicações a respeito de sua API. Com a introspecção isso não é mais necessário. Uma aplicação pode utilizar-se de introspecção [17, 24, 25] para detectar os métodos e variáveis disponíveis em uma classe e produzir seus nomes sem ter acesso ao código fonte. Isso só é possível porque o código objeto é aberto e examinado no ambiente de execução.

Se permitido pela política de segurança, através de introspecção pode-se: obter informações sobre um dado membro ou construtor, obter e modificar o valor dos campos, invocar métodos em objetos e classes, criar novas instâncias de uma classe.

Observa-se que, no caso dos agentes, a introspecção pode tornar-se bastante útil. Através dela, um agente pode descobrir os métodos de outros agentes e assim estabelecer uma comunicação.

A introspecção nos *beans* ocorre de forma mais direta, pois eles trabalham com padrões de projeto e interfaces, o que fornece uma maneira uniforme de fazer introspecção em diferentes *beans*.

3.4.4 *Class Loading*

O mecanismo de *class loading* [21] permite à máquina virtual carregar e definir classes em tempo de execução. Para carregar dinamicamente o código das aplicações e as classes referenciadas por elas existem as seguintes opções:

- uma aplicação serializada pode incluir suas classes, assim como qualquer classe referenciada;
- as classes de uma aplicação podem ser carregadas a partir de um servidor de *web* ou de algum outro servidor;
- as classes de uma aplicação podem ser carregadas através do *CLASSPATH*. O *CLASSPATH* é uma variável de ambiente que contém os diretórios e/ou os repositórios JARs (seção 5.4.2.1) onde devem se encontrar as classes Java a serem carregadas.

3.5 Considerações Finais

Neste capítulo foram apresentadas as principais características dos agentes de *software*. Não é intenção ressaltar nenhuma das características específicas dos agentes, pois a Interface Básica para um Servidor Universal (IBSU) tem como objetivo receber qualquer tipo de agente, verificar sua procedência e, dependendo da autenticidade do agente, permitir que ele execute sob as restrições impostas pelo *Security Manager* e pelas regras de controle de acesso aos *roots.*

Foi apresentada também a linguagem de programação Java. De acordo com o exposto, pode-se considerar Java como uma linguagem adequada para o desenvolvimento de agentes.

Com relação a Java, foi apresentada, primeiramente, a tecnologia Java *Beans* para o desenvolvimento de componentes, conceito que se encaixa em um dos principais objetivos dos agentes. Os *beans* também se mostram adequados para a construção de agentes que necessitam consultar bancos de dados. A característica de serialização, além de fornecer persistência, pode facilitar a implementação de agentes móveis, com capacidade de transportar-se de uma máquina a outra. O mecanismo de introspeção, através do qual uma

classe descobre as variáveis e métodos de outras classes, é uma facilidade de Java que desenvolve a capacidade social nos agentes.

O *class loader* permite que as classes que compõem uma aplicação (agente ou não) possam ser carregadas em tempo de execução. A segurança de Java impõe restrições para as classes carregadas, impedindo que aplicações não confiáveis tenham liberdade irrestrita. O capítulo 5 apresenta os motivos pelos quais a arquitetura de segurança de Java a torna segura para a hospedagem de agentes desconhecidos.

Capítulo 4

Mecanismos de Segurança

4.1 Considerações Iniciais

A segurança está relacionada à necessidade de proteção contra o acesso ou manipulação, intencional ou não, de informações confidenciais por elementos não autorizados. A necessidade de proteção deve ser definida em termos das possíveis ameaças e métodos de ataque, e formalizados nos termos de uma política de segurança [1].

Uma política de segurança é um conjunto de leis, regras e práticas que regulam como uma organização gerencia, protege e distribui suas informações e recursos. Um dado sistema é considerado seguro em relação a uma política de segurança, caso garanta o cumprimento das leis, regras e práticas definidas nessa política.

Neste capítulo são apresentados os principais métodos de ataque, as propriedades de segurança que devem ser levadas em consideração durante o desenvolvimento de um sistema seguro e os mecanismos de segurança existentes para promover as propriedades de segurança e assim, evitar os ataques.

4.2 Ameaças e Métodos de Ataque

Uma ameaça consiste em uma possível violação da segurança de um sistema. Algumas das principais ameaças às redes de computadores são:

- destruição de informação ou de outros recursos;
- modificação ou deturpação da informação;
- roubo, remoção ou perda de informação ou de outros recursos;
- revelação de informação sigilosa;
- interrupção de serviços.

As ameaças podem ser classificadas como acidentais ou intencionais, podendo ambas serem ativas ou passivas. Ameaças acidentais são as que não estão associadas à intenção premeditada (descuidos operacionais, *bugs* de *software* ou *hardware*). As ameaças intencionais são aquelas que configuram um ataque. A concretização das ameaças intencionais varia desde a observação de dados com ferramentas simples de monitoramento de redes, a ataques sofisticados baseados no conhecimento do funcionamento do sistema [26].

Ameaças passivas são as que, quando realizadas, não resultam em qualquer modificação no sistema. A realização de uma ameaça ativa envolve a alteração de informações contidas no sistema, ou em seu estado operacional.

Para violar um sistema de algum dos modos citados é necessário acessá-lo através dos canais de comunicação para acesso autorizado às facilidades de tal sistema. É através desses canais, que o acesso não autorizado também ocorre. De acordo com Coulouris et al. [1], os métodos de ataque podem ser dos seguintes tipos:

- **Espreita**: consiste na obtenção de cópias de mensagens sem autorização. É feito obtendo-se as mensagens diretamente da rede ou examinando informações que estão armazenadas sem proteção adequada.
- **Personificação**: enviar ou receber mensagens utilizando a identidade de outra pessoa ou processo (cliente, servidor, membro de um grupo).

- **Falsificação de mensagem**: consiste em interceptar mensagens e alterar seu conteúdo antes de enviá-las ao destino. Isso não é difícil de ser feito em redes *store-and-forward*.

- ***Replaying***: uma mensagem é interceptada e posteriormente transmitida, com o objetivo de produzir um efeito não autorizado.

- **Vírus**: é um programa que é acoplado a um programa legítimo do *host* e se instala em um ambiente sempre que esse programa é executado. Uma vez instalado, ele executa ações malignas, geralmente utilizando uma data como gatilho. Como o nome já diz, um dos seus ataques é se replicar e acoplar a todos os programas que puder encontrar no ambiente.

- **Infiltração**: ocorre quando um programa explora, remotamente, as facilidades de processos em execução.

- **Cavalo de Tróia**: consiste em um programa que é oferecido aos usuários de um sistema para executar uma função útil, mas que tem uma segunda função escondida. O exemplo mais comum é o *spoof login*, que apresenta aos usuários *prompts* que são iguais aos convencionais *login* e *password*, mas, na verdade, armazena as entradas do usuário em um arquivo para posterior uso ilícito.

4.3 Propriedades de Segurança

As propriedades de segurança de um sistema estão relacionadas à capacidade das entidades acessarem suas informações e recursos (bancos de dados, processadores, impressoras. etc.) de forma segura. Deve-se entender por entidades o conjunto de pessoas, chaves, processos ou máquinas.

De acordo com Chin [27], as propriedades que devem ser consideradas no desenvolvimento de um sistema seguro são:

- **Confidencialidade**: apenas as entidades envolvidas podem ter acesso ao conteúdo dos dados que estão trafegando na rede. Qualquer ação de monitoramento da rede não deve ser capaz de ter acesso dos dados.

- **Integridade**: deve-se garantir que a informação transmitida em um ponto é a mesma recebida em outro e que não houve nenhuma adulteração dos dados por parte de terceiros ou de falhas.

- **Autenticação**: as entidades envolvidas em uma comunicação devem ter meios de confirmarem mutuamente suas identidades, certificando-se de com quem estão se comunicando.

- **Controle de Acesso**: serve para restringir o acesso aos recursos. Assim, apenas entidades autorizadas poderão acessá-los.

- **Não Repudiação**: previne que entidades neguem suas ações. Assim, se uma entidade enviou uma mensagem, ela não poderá negar que o fez.

- **Disponibilidade de Serviços**: garante que entidades autorizadas acessem determinados serviços.

Os mecanismos de segurança servem para implementar as propriedades de segurança citadas. Eles são: criptografia, assinatura digital e autorização. As seções 4.4 a 4.6 abordam os mecanismos de segurança.

4.4 Criptografia

Até os anos 80, os maiores usuários da criptografia [1, 30] eram as instituições governamentais e os bancos. Hoje, os mecanismos de criptografia podem ser encontrados em muitas áreas de nossas vidas, onde eles são utilizados para evitar que informações sejam comprometidas ou alteradas sem autorização. Com o aumento da utilização da comunicação eletrônica para acesso público e o crescimento da Internet, a necessidade da criptografia tende a crescer cada vez mais.

A criptografia consiste em modificar a mensagem a ser transmitida, gerando uma mensagem criptografada na origem, através de um processo de codificação definido por um método de criptografia. A mensagem criptografada é então transmitida e, no destino, o processo inverso ocorre, isto é, o método de criptografia é aplicado novamente para decodificar a mensagem. A figura 4.1 mostra o funcionamento de um método de criptografia padrão.

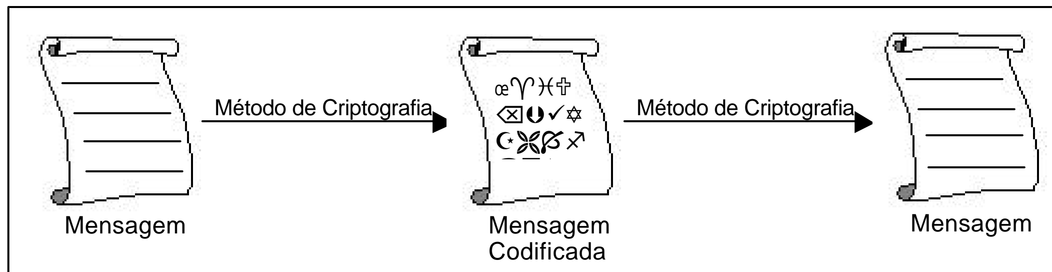

Figura 4.1 - Característica padrão de um método de criptografia

Utilizando-se esse método padrão, sempre que um intruso conseguisse descobrir o método de criptografia utilizado seria necessário substituí-lo.

Esse problema pode ser contornado através da utilização de uma chave. Esse modelo é ilustrado na figura 4.2, onde a mensagem criptografada varia de acordo com a chave de codificação utilizada para o mesmo método de criptografia. Isto é, para a mesma mensagem e um mesmo método de criptografia, chaves diferentes produzem mensagens criptografadas diferentes. Assim, o fato de um intruso conhecer o método de criptografia não é suficiente para que ele possa recuperar a mensagem original, pois é necessário fornecer ao procedimento responsável pela decodificação tanto a mensagem criptografada quanto a chave de decodificação.

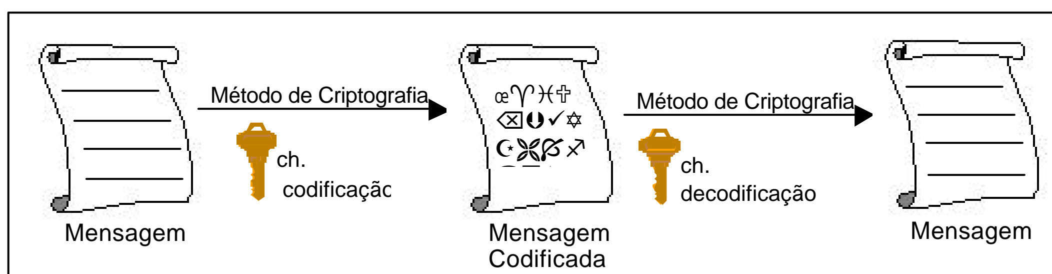

Figura 4.2 - Criptografia com chave para codificação e decodificação

Um bom método de criptografia deve garantir que seja, senão impossível, pelo menos muito difícil que um intruso recupere, a partir da mensagem criptografada e do conhecimento sobre o método de criptografia, o valor das chaves.

A criptografia é utilizada para garantir a propriedade de confidencialidade, mas não garante a autenticidade do emissor da mensagem, ou seja, utilizado-se apenas criptografia, não é possível saber a identidade de quem enviou a mensagem. Para isso, deve-se utilizar a criptografia juntamente com as assinaturas digitais, como descrito na seção 4.5.

A criptografia pode utilizar-se de chave secreta ou pública para codificar uma mensagem, como descrevem as subseções a seguir.

4.4.1 Criptografia com Chave Secreta

Na criptografia com chave secreta [1], a mensagem é criptografada aplicando-se uma função à mensagem com uma chave secreta. A decodificação é feita aplicando-se a função inversa ao texto criptografado, utilizando a mesma chave, para produzir a mensagem original. Como a chave é mantida em segredo, as funções de codificação e decodificação não precisam ser secretas.

Ambas as partes envolvidas na comunicação (emissor e receptor) devem possuir a função de criptografia e uma chave secreta. Antes que a comunicação comece, a chave secreta deve ser adquirida por ambos através de um canal seguro.

Na criptografia com chave secreta é necessário que haja confiança entre o emissor e o receptor, pois ambos devem ter a posse da chave secreta.

Um dos principais métodos de criptografia baseado em chave secreta é o DES [28] (*Data Encryption Standard*).

4.4.2 Criptografia Com Chave Pública

Na criptografia com chave pública não há necessidade de confiança entre o emissor e o receptor da mensagem, ao contrário da criptografia com chave secreta.

Cada receptor potencial de uma mensagem faz um par de chaves, C_e e C_d e mantém a chave de decodificação (C_d) em segredo. A chave de codificação C_e pode ser pública e utilizada por qualquer um que queira se comunicar. O método é baseado na utilização de uma função para definir a relação entre as duas chaves, assim, é muito difícil determinar o valor de C_d conhecendo-se C_e. Por exemplo, se uma pessoa P espera receber informações secretas de

outras pessoas, P gera um par de chaves C_e e C_d, disponibiliza C_e e mantém C_d em segredo. Isso pode ser feito mandando-se C_e diretamente para as pessoas de quem P espera receber informações ou para um serviço de distribuição de chave pública, que mantém um banco de dados de chaves públicas e as fornece para qualquer requisitante. Para enviar informações secretas para P, deve-se adquirir a chave pública de P, criptografar o texto utilizando a chave e enviá-lo para P. Apenas P conhece a chave C_d e a utiliza para decodificar o texto.

O mais importante método de criptografia com chave pública é o RSA [29], cujo nome deriva das iniciais dos autores Rivest, Shamir e Adleman.

4.5 Assinatura Digital

A criptografia é utilizada para implementar o mecanismo de assinatura digital [1, 31]. A função das assinaturas digitais no mundo eletrônico é o mesmo das assinaturas no papel do mundo real. Desde que uma chave privada é conhecida apenas pelo seu possuidor, a utilização dessa chave é vista como uma evidência de identidade. Assim, se uma mensagem for criptografada com a chave privada de um usuário, pode ser deduzido que a mensagem foi "assinada" diretamente pelo usuário.

O mecanismo de assinatura digital envolve dois procedimentos: assinatura de uma mensagem e verificação dessa assinatura. Uma mensagem M pode ser assinada por uma entidade P através da codificação de uma cópia de M com uma chave C_a (única e secreta) pertencente a P, acoplando-se isso ao texto original de M e ao identificador de P. Assim, um documento assinado consiste de $<M, P, \{M\}_{Ca}>$. O propósito de se acoplar uma assinatura a um documento é de permitir que qualquer um que receba o documento possa verificar que ele se originou de P e que o conteúdo de M não foi modificado. A verificação da assinatura pode ser feita através de chave secreta ou chave pública.

As assinaturas digitais são usualmente utilizadas em conjunto com as funções *hash* e certificados, como descreve as duas subseções seguintes. Quando utilizadas nessas condições, as assinaturas digitais servem para garantir as propriedades de integridade, autenticação, controle de acesso e não repudiação [27]. Para garantir também a confidencialidade, a mensagem M, que está sendo enviada juntamente com $\{M_{Ca}\}$ e P, deve estar criptografada.

4.5.1 Função *Hash*

Os algoritmos de *hash* são utilizados para produzir a "impressão digital" dos dados, ou seja, identificadores de dados únicos e confiáveis. Esses algoritmos obtém uma entrada de tamanho arbitrário e geram uma saída de tamanho fixo, chamada *hash* ou *digest*, que tem as seguintes propriedades:

- Deve ser computacionalmente impossível achar duas mensagens que originem o mesmo valor.
- O *hash* não revela nada sobre a entrada.

As funções *hash* são utilizadas freqüentemente em conjunto com assinaturas digitais, com o objetivo de identificar unicamente uma mensagem. As figuras 4.3 e 4.4, respectivamente, ilustram o processo de geração e verificação de uma assinatura digital juntamente com criptografia.

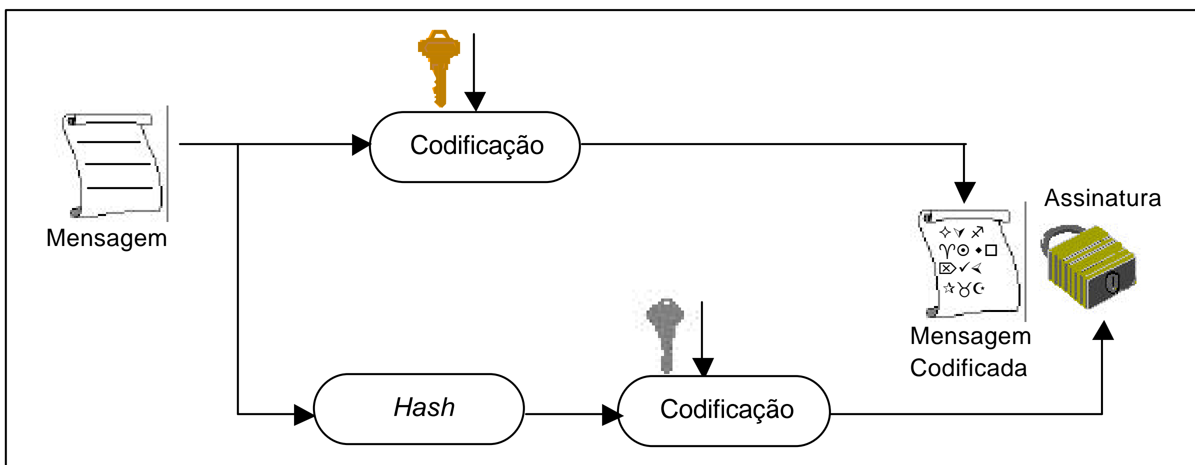

Figura 4.3 - Geração de assinatura digital

De acordo com a figura 4.3, o valor *hash* da mensagem é gerado na origem. Ao valor do *hash* é aplicado um método de criptografia com uma chave privada pertencente ao assinante, o que é uma evidência de identidade. Assim é gerada a mensagem assinada ou simplesmente assinatura. Deve-se enviar uma cópia da mensagem original juntamente com a assinatura, para que elas possam ser comparadas no destino. Para garantir a confidencialidade da mensagem, deve-se codificá-la através de um método de criptografia. Dessa forma, se a mensagem for interceptada, seu conteúdo não poderá ser lido.

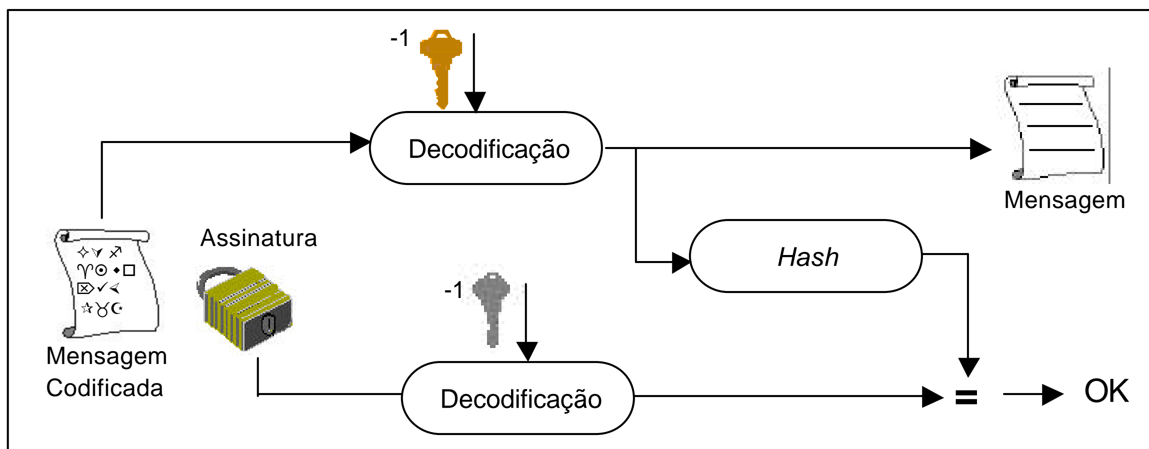

Figura 4.4 – Verificação de assinatura digital

A assinatura é enviada pela rede juntamente com a mensagem codificada. No destino, acontece o processo inverso, como ilustrado na figura 4.4. A mensagem assinada é decodificada utilizando-se o mesmo método de criptografia da origem e a chave de decodificação, que pode ter sido enviada juntamente com a mensagem ou obtida através de um serviço de distribuição de chaves públicas. A mensagem codificada também sofrerá o processo inverso para obter-se a mensagem original. Um novo valor *hash* é gerado para a mensagem original e comparado ao que foi decodificado a partir da assinatura. Se os *hashes* forem iguais significa que o conteúdo da mensagem original não foi alterado e que o possuidor da chave privada é o único que pode ter enviado a mensagem.

Como exemplos de algoritmos de *hash* pode-se citar o SHA [32] (*Secure Hash Algorithm*), o MD2 [33] (*Message Digest*) e o MD5 [34].

Para as assinaturas digitais utiliza-se um algoritmo de *hash* combinado com um algoritmo que implemente um mecanismo de criptografia, tais como: MD2 com RSA, MD5 com RSA e SHA1 com RSA.

4.5.2 Certificação

Considerando a figura 4.4, caso os valores *hashes* sejam iguais, surge uma questão crucial: quem é o possuidor da chave privada? Uma mensagem só pode ser autenticada caso

exista a certeza de que a chave pública recebida da entidade que enviou a mensagem é realmente a chave pública dessa entidade. Como ter certeza que alguém não enviou a chave pública no lugar do verdadeiro possuidor dela? Essas questões geram as seguintes implicações [27]:

- Qual é o critério para a associação de chaves a entidades?
- Quais são as autoridades que garantem a autenticidade das entidades, as informações a respeito delas e seus privilégios?

A resposta para todas essas questões está em certificados e autoridades certificadoras [27]. Os certificados servem para documentar a associação de chaves públicas com entidades. São declarações digitalmente assinadas por um possuidor de chave privada autorizado, dizendo que a chave pública de uma determinada entidade é autêntica. Um certificado consiste também de informações detalhadas sobre o dono da chave pública (nome, endereço, etc.).

Para garantir a integridade do certificado, ele é assinado por uma autoridade certificadora, ou seja, uma entidade confiável, cuja chave pública é amplamente divulgada. A autoridade certificadora serve para garantir que a identidade do possuidor do certificado corresponde realmente a sua pessoa (ou entidade). Desse modo, a autoridade certificadora garante a validade da chave pública contida no certificado.

O certificado é assinado pela autoridade certificadora utilizando a chave privada dessa autoridade. Para obter as informações do certificado deve-se decodificá-lo com a chave pública da autoridade certificadora que o assinou.

Existem também os chamados certificados *self-signed*. Nesse tipo de certificado, não há a presença de uma autoridade certificadora. Se uma pessoa ou entidade possui um par de chaves, ela utiliza sua própria chave privada para assinar o certificado da chave pública. Ela mesma diz que o certificado da sua chave pública é autêntico. Isso é adequado apenas se a entidade que irá receber a mensagem assinada conhecer e confiar na identidade de quem enviou.

Um dos padrões mais utilizados de certificados é o X.509 [8], adotado pelas autoridades certificadoras *Microsoft's Authenticode*, *Netscape's Object Signing* e *Marimba's Channel Signing* para autenticar a origem de objetos da Internet.

4.6 Autorização

Os mecanismos de autorização estão diretamente relacionados ao controle de acesso [1], pois servem para garantir que o acesso a recursos de informação (como arquivos, processos ou portas de comunicação) e de *hardware* (como servidores de impressão) só seja permitido para o conjunto de usuários autorizado a acessá-los.

É de responsabilidade da aplicação implementar mecanismos de autorização para acesso aos seus recursos. No UNIX e em outros sistemas multiusuários, por exemplo, os arquivos são as informações compartilhadas mais importantes e um esquema para autorização de acesso é oferecido para permitir a cada usuário manter arquivos privados e poder compartilhá-los de forma controlada.

A proteção de recursos é específica do serviço. Os *kernels*, por exemplo, implementam sua própria proteção de recursos e os serviços de alto nível fazem o mesmo. *Kernels* fornecem facilidades de *hardware* tais como unidades de gerenciamento de memória para implementar proteção para si próprios e para o *hardware* que eles gerenciam, assim como proteção de memória para seus processos. Os servidores, por outro lado, têm que estar atentos ao fato de poderem receber mensagens de qualquer lugar do sistema. Eles devem ser implementados de forma a proteger seus recursos contra requisições maliciosas.

4.7 Considerações Finais

A abertura dos sistemas distribuídos os expõe a diversas ameaças à segurança. Como foi exposto neste capítulo, existem mecanismos de segurança para proteção de sistemas distribuídos. Esses mecanismos são baseadas no uso da criptografia, não apenas para encobrir informação, mas também para autenticá-la.

Cabe mencionar que a segurança pode ser atingida levando-se em consideração a confiança existente entre as entidades envolvidas e o meio de comunicação, o que acarreta três possibilidades:

- As entidades e os meios de comunicação são todos confiáveis. Nesse caso, pode-se obter segurança através de uma senha.

- Cada entidade confia em seu parceiro, porém não confia no meio de comunicação. Nesse caso a segurança pode ser fornecida com o emprego de métodos de criptografia.

- As entidades não confiam nos seus parceiros (ou sentem que não poderão confiar no futuro) nem no meio de comunicação. Nesse caso, devem ser utilizadas técnicas que impeçam que uma entidade negue que enviou ou recebeu uma mensagem, ou seja, mecanismos de assinatura digital.

A Interface Básica para um Servidor Universal (IBSU) considera a terceira possibilidade, ou seja, o meio de comunicação não é seguro e as entidades não confiam em seus parceiros, o que implica na utilização de assinatura digital. Na IBSU a checagem de assinatura digital e certificado(s) é feita pelo agente *Gateway* utilizando-se recursos de segurança da linguagem Java, como detalhado nos capítulos 5 e 6.

A autorização se dá em nível de acesso aos métodos disponibilizados pela Interface com o Banco de Dados e de utilização de determinados recursos de Java. Na IBSU, o controle de acesso é feito pelos métodos definidos na Interface com o Banco de Dados e pelo *SecurityManager*, como detalhado no capítulo 6.

Capítulo 5

A Segurança em Java

5.1 Considerações Iniciais

Para Li Gong [35], de um ponto de vista industrial, um plataforma é considerada confiável caso ela atenda aos seguintes requisitos:

- **Usabilidade**: para ser aceita no mercado, a plataforma deve oferecer facilidades para construção de sistemas e aplicações de qualquer porte.
- **Simplicidade**: para inspirar confiança, a plataforma não pode ser muito complexa.
- **Suficiência**: a plataforma deve conter todas as características necessárias para suportar requisitos de segurança.
- **Adaptabilidade**: a plataforma deve evoluir com facilidade, de acordo com a demanda e a realidade do mercado.

Considerando-se essas características, a linguagem de programação Java mostra-se bastante favorável quando comparada a outras linguagens. A portabilidade de Java reduz a complexidade de lidar-se com ambientes heterogêneos e assim a probabilidade de erros de projeto e implementação. As características de segurança existentes em Java contribuem para sua usabilidade, simplicidade, suficiência e adaptabilidade no cenário global da computação.

Este capítulo apresenta a segurança em Java considerando todos os seus aspectos, o que inclui a própria linguagem, as restrições impostas pela máquina virtual, a API e as ferramentas que possibilitam a implementação dos mecanismos de segurança, descritos no capítulo 4.

5.2 Linguagem Segura

Java possibilita o desenvolvimento de programas seguros. Isso significa que Java previne falhas acidentais nos programas. Os principais recursos da linguagem que proporcionam tal conforto são [36]:

- **Endereçamento automático de ponteiros**: a linguagem Java não permite o uso de ponteiros por parte do programador. Eles existem, mas o programador não os manipula diretamente. Esse talvez seja o recurso mais útil na prevenção à falhas de programas e máquinas, pois áreas de memória não apropriadas não serão acessadas.

- ***Garbage Collection***: esse recurso permite que o ambiente de execução de Java (JRE) possa liberar automaticamente o espaço de memória que não estiver mais sendo referenciado. Com o *garbage collector*, o programador não necessita saber quando, ou se é seguro, liberar um espaço de memória.

- **Controle de *casting***: a verificação de tipos é feita em tempo de compilação e execução. Java não permite que um bloco de memória seja interpretado como sendo de um tipo diferente do que ele realmente é. Por exemplo, não se pode fazer o *casting* de um *array* de `Object` para um *array* de `String`, a não ser que o *array* de `Object` seja do tipo `String`.

- **Modificadores de acesso**: os membros das classes em Java podem ser definidos como públicos, protegidos, *friendly* e privados. Assim, pode-se restringir o acesso aos membros de uma classe.

5.3 Segurança na Máquina Virtual

As aplicações Java necessitam da máquina virtual para poderem executar. A *Java Virtual Machine* (JVM) conta com os recursos descritos a seguir para garantir a execução segura das aplicações [36, 37]:

- **Verificação de *byte codes***: quando uma classe Java é carregada, antes de mais nada, a JVM verifica a validade dos *byte codes* do arquivo que foi carregado. Os *byte codes* são a linguagem de máquina da JVM, como descrito na sua especificação [38]. O compilador Java gera os *byte codes* (.class) a partir de um arquivo fonte, como mostra a figura 3.1. A verificação de *byte codes* também controla o acesso aos membros, a verificação de super classes e de argumentos passados como parâmetro aos métodos.

- ***Class Loading***: depois que os *byte codes* foram verificados, a classe *ClassLoader* entra em ação. É o *ClassLoader* que procura e carrega os *byte codes* a partir da definição das classes. Além disso, o *ClassLoader* impede que classes da API de Java sejam carregadas via rede, o que garante que a JVM não está utilizando falsas representações das bibliotecas Java. O *ClassLoader* fornece também espaços de nome separados para classes carregadas de diferentes localidades, garantindo que classes com o mesmo nome, vindas de *hosts* diferentes, não entrarão em conflito.

- ***SecurityManager***: provavelmente o aspecto de segurança mais importante de Java é de responsabilidade da classe *SecurityManager*. Cada JVM executando tem no máximo um *SecurityManager* instalado. Sempre que uma aplicação tentar realizar uma operação que pode ser prejudicial ao sistema (escrever no sistema de arquivos, por exemplo), o *SecurityManager* irá verificar se aquela operação é permitida para a aplicação em questão. No modelo de segurança da versão Java 1.2x o *SecurityManager* permite o estabelecimento de uma política de segurança de forma relativamente simples, assim pode-se restringir ou liberar recursos para uma determinada aplicação Java. Mas, o modelo de segurança da plataforma Java nem sempre ofereceu tal facilidade. A evolução do modelo de segurança de Java é apresentada a seguir. A definição de uma política de segurança será melhor abordada na seção 5.5.

A Evolução do Modelo de Segurança de Java

O modelo de segurança original da plataforma Java é conhecido como modelo *sandbox*[39,40]. O objetivo do *sandbox* era fornecer um ambiente muito restrito, no qual aplicações não confiáveis obtidas através da rede seriam executadas. A essência do *sandbox* é que o código local é confiável e por isso tem acesso total aos recursos do sistema (arquivos, conexões de rede, etc.), enquanto o código carregado via rede (uma *applet*) não é confiável e pode acessar apenas os recursos limitados disponíveis dentro do *sandbox*. *Applets* que executam sob as restrições do *sandbox* não podem ler ou escrever em arquivos do sistema local e nem criar conexões com outros *hosts*, a não ser aquele de onde a *applet* foi carregada. O modelo *sandbox* foi proposto como *default* no *Java Development Kit* (JDK) 1.0 e foi amplamente adotado por aplicações construídas com o JDK 1.0, o que inclui *browsers* com suporte à Java. A figura 5.1 ilustra o modelo *sandbox*.

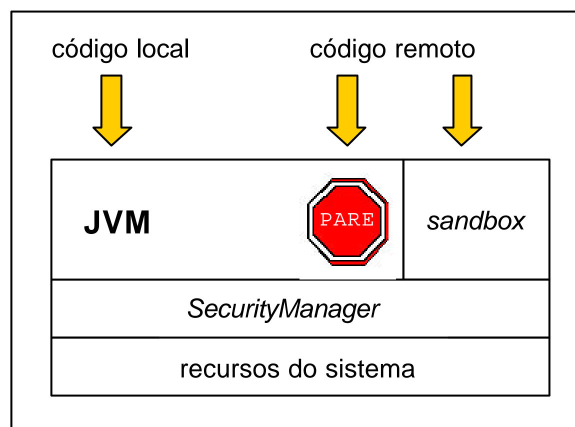

Figura 5.1 - Modelo de segurança do JDK 1.0

Devido às limitações do modelo de segurança do JDK 1.0, seria impossível, por exemplo, a implementação de uma *applet* para redação de textos, que fosse capaz de salvar arquivos no disco local.

A discriminação contra as *applets* teve seu fim com o modelo de segurança do JDK 1.1. Com ele foram introduzidas as assinaturas digitais (seção 4.5) para as *applets*. Como ilustra a figura 5.2, um *applet* devidamente assinado era considerado confiável e tratado como código local.

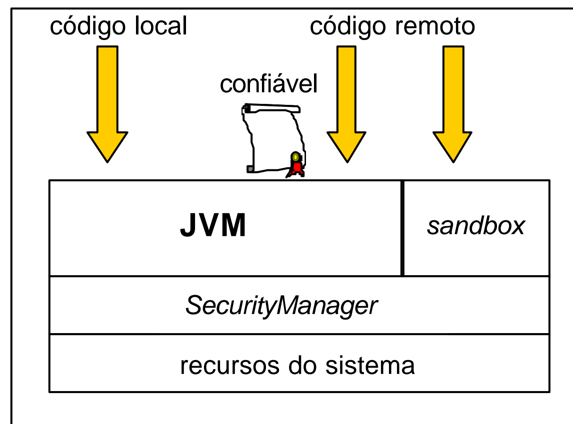

Figura 5.2 - Modelo de segurança do JDK 1.1

O modelo de segurança *default* do JDK 1.1 resolveu parcialmente o problema, pois não existia nenhuma granulosidade. A execução do código remoto era feita sob dois extremos. Ou a *applet* executaria no *sandbox* ou teria acesso irrestrito a todos os recursos do sistema. Até então, se uma aplicação ou *applet* desejasse ter uma política de segurança diferente, ela teria que criar subclasses das classes *SecurityManager* e *ClassLoader*, e personalizá-las de acordo com sua necessidade. Entretanto, tal prática requer profundo conhecimento de segurança e pode tornar-se perigosa por envolver uma questão sensível como essa [40].

A solução veio com o modelo de segurança de Java 2, introduzido com o JDK 1.2. Com o novo modelo, uma aplicação pode possuir um domínio de proteção, que é formado por um conjunto de permissões. Como ilustra a figura 5.3, é possível estabelecer uma política de segurança para a aplicação a ser executada. É a política de segurança que irá definir o domínio de proteção para uma aplicação, independente dela ser local ou remota, assinada ou não. Assim, o código em questão, seja ele aplicação ou *applet*, *bean* ou *servlet*, pode ter um conjunto de permissões específicas. A seção 5.5 descreve com que recursos Java 2 possibilita o estabelecimento de uma política de segurança.

A Interface Básica para um Servidor Universal (IBSU) utiliza-se do modelo de segurança do JDK 1.2 devido a sua maior flexibilidade e granulosidade. As seções seguintes apresentam os recursos que Java 2 oferece para o desenvolvimento de aplicações que utilizam assinaturas digitais e mecanismos de autorização.

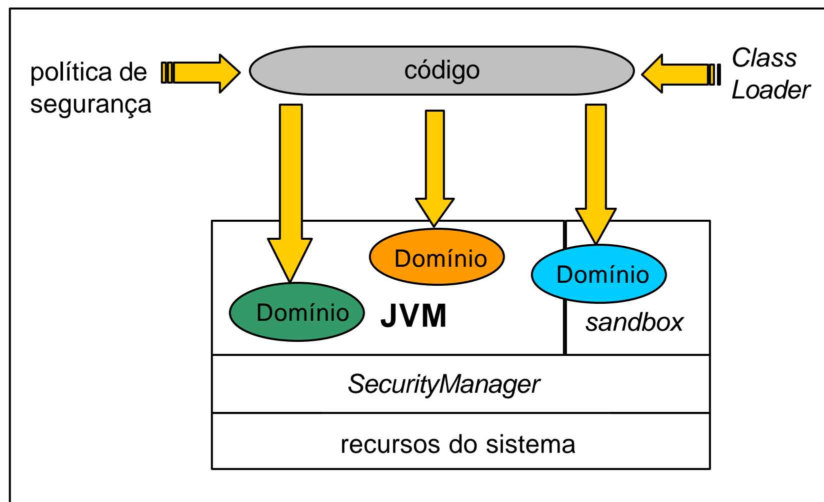

Figura 5.3 - Modelo de segurança do JDK 1.2

5.4 Suporte a Assinaturas Digitais

Como visto no capítulo anterior, a geração, verificação e autenticação de assinaturas digitais envolve os seguintes processos:

- geração de um par de chaves;
- geração do valor *hash* da mensagem a ser assinada;
- assinatura da mensagem, a partir do valor *hash*, utilizando-se a chave privada;
- decodificação da mensagem;
- geração de um novo valor *hash* para a mensagem;
- comparação do novo *hash* com o gerado na origem, os quais devem ser idênticos se a mensagem não sofreu alterações em trânsito;
- verificação do certificado do assinante da mensagem, o que garante sua autenticidade.

Para fornecer o mecanismo de assinatura digital, Java 2 conta com uma API de segurança e com as ferramentas *keytool* e *jarsigner*.

5.4.1 A API de Segurança de Java 2

A plataforma Java 2 fornece uma API de segurança [35, 41] que possibilita a assinatura e verificação da assinatura de dados. A API fornece a implementação dos algoritmos SHA1 com DSA, MD2 com RSA, MD5 com RSA e SHA1 com RSA para assinaturas digitais, o que não significa que o desenvolvedor está limitado a essas opções.

A API é composta pelos pacotes `java.security`, `java.security.acl`, `java.security.interfaces`, `java.security.cert` e `java.security.specs`. Através deles pode-se:

- registrar a implementação de novos algoritmos e serviços de segurança;
- criar pares de chaves;
- criar valores *hash* dos dados a serem assinados;
- gerar a assinatura para um dado utilizando a chave privada;
- armazenar e recuperar de arquivos a chave pública e a assinatura;
- verificar a assinatura de um dado utilizando a chave pública;
- acessar e modificar informações armazenadas em *keystores* (arquivos protegidos por senha onde a ferramenta *keytool*, descrita na seção 5.4.2.2, armazena chaves e certificados);
- gerar certificados a partir do valor codificado desses certificados, usualmente obtidos de *keystores*.

A API não possibilita a criação de certificados, ela apenas cria os caracteres ASCII para os certificados a partir de um valor codificado já existente para eles. Essa é uma limitação na implementação do mecanismo de assinatura digital via API, pois nada garante a autenticidade da chave pública [44]. Entretanto, a API possibilita que os certificados sejam recuperados de *keystores* criados pela ferramenta *keytool* (seção 5.4.2.2). Neste caso, utilizando-se a API em conjunto com a ferramenta, pode-se garantir a autenticidade da chave pública.

5.4.2 As Ferramentas Java 2 para Assinatura Digital

Além da API de segurança, Java 2 fornece ferramentas para a assinatura digital, o que possibilita a assinatura de *applets* e aplicações. A ferramenta *keytool* manipula chaves e certificados, enquanto a ferramenta *jarsigner* gera e verifica as assinaturas, que devem ser geradas a partir de uma entrada em um arquivo JAR.

5.4.2.1 Os Arquivos JAR

Um Java ARchieve ou simplesmente JAR [42, 43] é um arquivo com a extensão .jar, gerado com a ferramenta *jar* fornecida no JDK 1.2. As entradas de um arquivo JAR podem ser assinadas digitalmente pelo autor de uma *applet* ou aplicação.

JAR é baseado no popular formato de arquivo ZIP e é utilizado para compactar e agregar vários arquivos em um. Apesar de possibilitar assinatura digital, a motivação primordial para o desenvolvimento de arquivos JAR é que as *applets* e seus componentes, tais como arquivos de imagem e áudio, possam ser submetidos a *download* em apenas uma transação. Isso melhora grandemente a performance, pois não é necessário abrir uma nova conexão para cada arquivo.

Ao criar-se um JAR é criado também um arquivo *manifest*, chamado, por *default*, `META-INF/MANIFEST.MF`. Esse arquivo consiste de várias seções, cada uma delas é uma entrada correspondente a um arquivo que compõe o JAR. É a partir dessas entradas que a assinatura digital será gerada. O processo para geração da assinatura de um JAR é descrito na seção 5.4.2.3.

5.4.2.2 A ferramenta *keytool*

Keytool [42,44] é uma ferramenta para gerenciar chaves e certificados. Ela armazena chaves e certificados em *keystores*. Os *keystores* são arquivos onde a chave privada é protegida por senha.

Keytool permite que usuários administrem seus próprios pares de chaves e os certificados associados a eles. Eles podem também armazenar certificados de chaves públicas (que contém as chaves públicas) de outras entidades, com as quais desejam comunicar-se.

Podem existir dois tipos de entrada em um *keystore*: para as chaves, o que inclui a chave privada e o certificado contendo a chave pública de uma determinada entidade, e para certificados (contendo a cheve pública de outras entidades) confiáveis. A entrada para as chaves armazena as informações de forma secreta em um formato protegido, para prevenir acesso não autorizado. A outra entrada armazena certificados confiáveis. Um certificado é dito confiável porque o possuidor do *keystore* confia na chave pública daquele certificado. Essa entrada é necessária caso o possuidor do *keystore* deseje receber dados/aplicações de outras entidades e autenticá-las. Os dois tipos de entrada do *keystore* estão associadas a um *alias*. É o *alias* que identifica o possuidor de chaves e certificados confiáveis no *keystore*, como ilustra a figura 5.4.

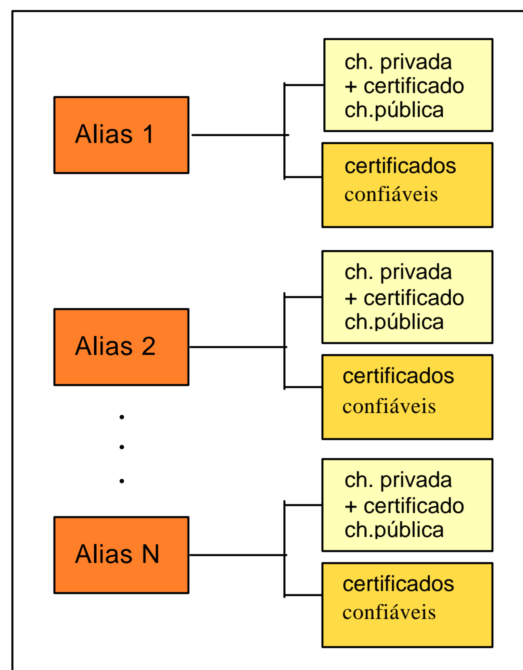

Figura 5.4 – Estruturação do *keystore*

Como pode ser observado, um *keystore* pode conter vários *alias*. O *alias* é um nome utilizado para identificar o possuidor de chaves e certificados no *keystore*. Por exemplo, se existe um *alias* chamado *superusers*, a primeira entrada armazena a chave privada e o certificado contendo a chave pública de *superusers*. Para acessar as chaves, deve-se fornecer o *alias*. A segunda entrada armazena certificados contendo a chave pública de outras entidades confiáveis para *superusers*, com as quais *superusers* provavelmente irá se comunicar no futuro.

Os certificados podem ser exportados e importados do *keystore*. Exportar um certificado significa extraí-lo do *keystore* para que ele possa ser enviado a uma entidade que precisará autenticar uma chave pública. A entidade que recebeu o certificado exportado deve, então, entrar em contato com o possuidor do certificado e verificar sua autenticidade comparando os *fingerprints* (ou *hashes*) do certificado original com os gerados a partir do certificado recebido. Se os *fingerprints* forem iguais significa que o certificado é válido, então basta importá-lo, ou seja, adicioná-lo à lista de certificados confiáveis do *keystore*.

A ferramenta *keytool* gera os certificados *self-signed*, ou seja, o próprio possuidor da chave privada garante a autenticidade do certificado contendo a chave pública (seção 4.5.2). Com esse tipo de certificado o ideal é que o receptor do JAR conheça e confie na identidade de quem o assinou. Caso não exista tal confiança, é melhor que o certificado seja assinado por uma entidade certificadora. *Keytool* possibilita a criação de uma requisição de assinatura para certificado, ou seja, gera-se um arquivo, que será enviado para uma entidade certificadora, contendo uma requisição de assinatura. Para gerar a requisição, já deve existir no *keystore* a chave privada e o certificado *self-signed*. A *Keytool* gera a requisição a partir desses dados armazenados. Estando de posse do arquivo para requisição, deve-se enviá-lo para uma entidade certificadora, que irá assinar o certificado e retorná-lo.

Tendo um certificado com a chave pública autenticada por uma autoridade certificadora, basta substituir o certificado *self-signed* do *keystore* pelo novo certificado. Os certificados gerados pela ferramenta *keytool* estão no padrão X.509. Mais informações a respeito de certificados *self-signed* e requisições para autoridades certificadoras podem ser obtidas em Dageforde [44].

5.4.2.3 A ferramenta *jarsigner*

A ferramenta *jarsigner* [42, 44] gera assinaturas digitais para um arquivo JAR e as verifica. A *Jarsigner* utiliza os algoritmos DSA com SHA1 ou RSA com MD5.

Para que a assinatura possa ser gerada deve existir primeiramente uma chave privada e o certificado da chave pública correspondente. A *Jarsigner* utiliza informações sobre chaves e certificados armazenadas em um *keystore* para gerar a assinatura. *Jarsigner* acessa uma determinada entrada no *keystore* a partir de um *alias*.

Um JAR assinado contém, entre outras coisas, o certificado extraído do *keystore* autenticando a chave pública correspondente à chave privada que foi utilizada para assiná-lo.

O arquivo JAR assinado é exatamente igual ao original, a não ser por dois arquivos que são acrescentados no diretório `META-INF/`: o arquivo *signature*, com extensão .SF, e o arquivo *signature block*, com a extensão .DSA.

O arquivo .SF é muito similar ao arquivo *manifest* (seção 5.4.2.1). Ambos contêm uma seção para cada arquivo do JAR. Cada seção contém três linhas: o nome do arquivo, o nome do algoritmo utilizado para gerar o valor *hash* para aquele arquivo e o valor *hash* para o arquivo. Essas três linhas são chamadas de entradas do arquivo.

A diferença é que, no *manifest*, o *hash* de cada arquivo é gerado a partir dos dados armazenados nele. No .SF o *hash* de cada arquivo é gerado a partir das três linhas do *manifest*. Esse esquema garante que o conteúdo dos arquivos do JAR não serão alterados.

O arquivo .SF é assinado e a assinatura é armazenada no arquivo .DSA. Além disso, o .DSA contém o certificado codificado, o qual serve para autenticar a chave pública correspondente à chave privada utilizada para gerar a assinatura. A figura 5.5 ilustra o processo para gerar a assinatura digital de um JAR.

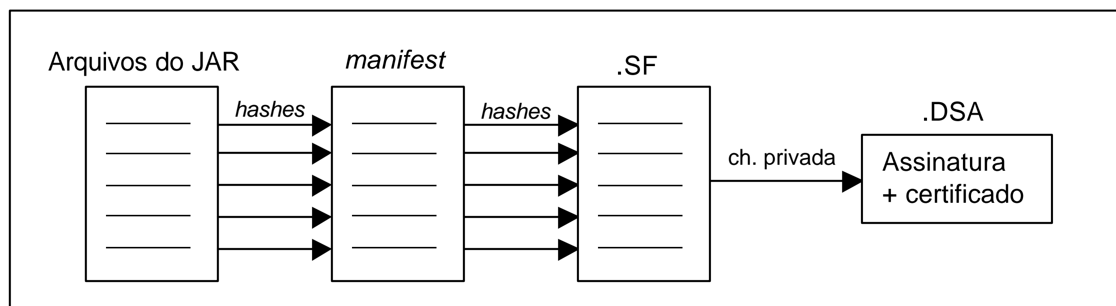

Figura 5.5 - Assinatura de um JAR

A verificação do JAR é feita com sucesso se a assinatura for válida e se nenhum dos arquivos que compõem o JAR foram alterados depois da assinatura ter sido gerada. A figura 5.6 ilustra o processo para verificação da assinatura digital de um JAR que envolve os seguintes passos:

1. Verificar a assinatura do arquivo .SF. Essa verificação assegura que a assinatura no arquivo .DSA foi gerada utilizando-se a chave privada correspondente à chave pública cujo certificado está armazenado no .DSA.

2. Verificar se os *hashes* de cada entrada no .SF são iguais aos *hashes* da entrada correspondente no *manifest*. Para isso é necessário gerar novos valores *hash* para as entradas do *manifest* e compará-las aos *hashes* existentes no .SF.

3. Ler cada arquivo do JAR que contém uma entrada no .SF e gerar um novo *hash* para eles. Se os *hashes* já existentes no arquivo *manifest* forem iguais aos novos valores, significa que os arquivos que compõem o JAR não foram modificados e a verificação é terminada com sucesso.

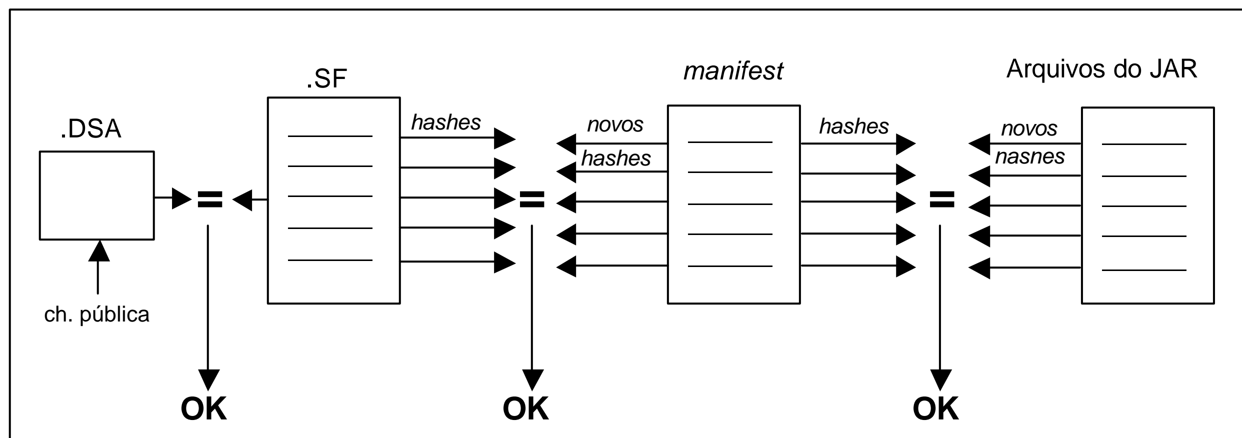

Figura 5.6 - Verificação da assinatura de um JAR

O processo de verificação da assinatura feito pela ferramenta *jarsigner* não inclui a verificação do certificado da chave pública. Para isso, uma cópia do certificado já deve ter sido importada para ser comparada ao certificado que chegou com o JAR. A API de Java 2 permite que o certificado seja extraído do JAR e comparado com os certificados armazenados em um *keystore*, o que garante a autenticidade da chave pública.

É importante observar que a assinatura de um arquivo JAR não garante sua confidencialidade. Ele pode ser capturado em trânsito e o conteúdo de seus arquivos pode ser

lido, porém não modificado. Uma solução é utilizar o *Java Cryptography Extension* (JCE), um pacote adicional para o JDK, que possui APIs para codificar e decodificar dados.

5.5 Definição de Políticas de Segurança

De acordo com a figura 5.3, com o novo modelo de segurança de Java 2 é possível estabelecer uma política de segurança para a aplicação a ser executada. Para isso, deve-se definir os domínios de proteção [40] para aquela aplicação. O *sandbox* é um exemplo de domínio de proteção.

Cada aplicação que irá executar no ambiente Java deve ter seu próprio domínio de proteção. Assim, uma aplicação pode ter, por exemplo, acesso ao sistema de arquivos, mas não ter acesso à rede. Isso significa que seu domínio de proteção é composto apenas pelo sistema de arquivos. A política atualmente em efeito no sistema é o conjunto dos domínios de proteção.

Os domínios de proteção são definidos através de permissões. Uma permissão representa o acesso a um recurso do sistema. Então, se uma aplicação executando com *SecurityManager* quiser acessar um determinado recurso, a permissão correspondente deve ser explicitamente concedida a ele. Os tipos de permissões controladas pelo ambiente Java são:

- `AWTPermission`: a biblioteca `java.AWT` é geralmente utilizada pelos *applets*, pois ela possibilita a manipulação de eventos a partir de uma interface gráfica. As permissões associadas a `AWTPermissons` controlam o acesso dos *applets* a determinados eventos.

- `FilePermission`: representa o acesso a arquivos e diretórios.

- `NetPermission`: controla permissões associadas à rede, o que inclui construção de URLs e informações para autenticação.

- `PropertyPermission`: controla o acesso à propriedades do sistema, como *usernames* e diretório *home* de usuários.

- `ReflectPermission`: diz respeito à operações que utilizam-se de introspecção (seção 3.4.3).

- `RuntimePermission`: define permissões para o acesso e modificação do *ClassLoader*, *SecurityManager*, operações com *threads*, carregamento dinâmico de bibliotecas, etc.

- `SecurityPermission`: controla as permissões relacionadas à política de segurança do sistema, chaves e certificados.

- `SerializablePermission`: refere-se à permissões relacionadas à serialização (seção 3.4.2)

- `SocketPermission`: representa o acesso à rede via *sockets*. Pode estabelecer a que *hosts* e portas uma determinada aplicação pode conectar-se.

- `AllPermission`: se for definido que uma aplicação tem `AllPermission`, significa que ela tem todas as outras permissões descritas acima.

Mais informações sobre os tipos de permissões e seus respectivos métodos para controle de acesso podem ser obtidas na *Sun Microsystems* [45].

A política para um ambiente Java é representada por um objeto *Policy*. Esse objeto pode ser especificado dentro de um ou mais arquivos de configuração de política ou *policy files*. São os *policy files* que especificam as permissões associadas a cada aplicação ou entidade, como descrito a seguir.

Os policy files

Um *policy file* [46] pode ser criado através de um editor simples ou através da ferramenta gráfica *policytool* [42, 44], incluída no JDK 1.2. Com a utilização dessa ferramenta não é necessário conhecer a sintaxe utilizada nos *policy files* (o que reduz grandemente a probabilidade de erros), somente que permissões serão concedidas e a quais entidades concedê-las.

A figura 5.7 ilustra os componentes de um *policy file* e o relacionamento entre eles. O *policy file* pode conter várias entradas, cada uma possui três propriedades:

- **Permissões**: é o conjunto das permissões definidas para a entrada em questão.

- ***CodeBase***: é opcional e indica a URL do código a ser executado. Se a entrada estiver associada a um *CodeBase* significa que apenas ao código originado daquela URL serão concedidas as permissões, caso contrário, as permissões serão concedidas ao código de qualquer URL.

- ***SignedBy***: é opcional e indica o *alias* do *keystore* onde está armazenado o certificado da chave pública correspondente à chave privada que assinou o código a ser executado. É por isso que um *policy file* geralmente está associado a um *keystore*. Se a entrada estiver associada ao campo *SignedBy*, apenas ao código assinado pelo *alias* correspondente serão concedidas as permissões, caso contrário, elas serão concedidas a qualquer código sem assinatura.

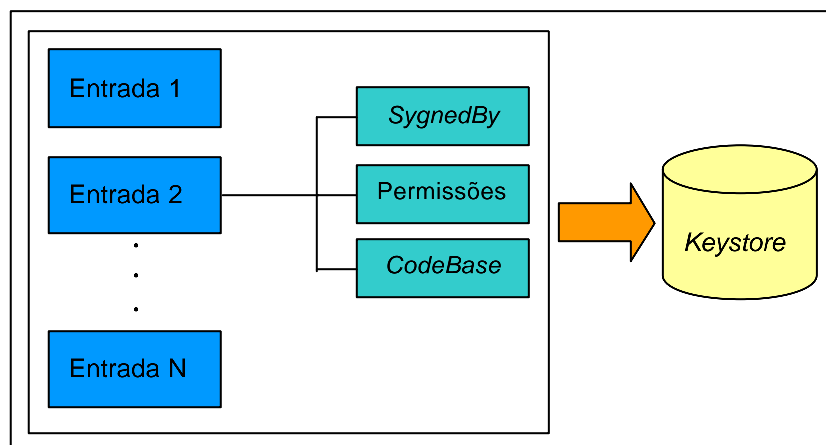

Figura 5.7 – Estruturação dos *policy files*

5.6 Considerações Finais

Através do exposto neste capítulo, pode-se concluir que a linguagem Java fornece todos os recursos para garantir a segurança de aplicações.

A figura 5.8 ilustra as ferramentas de Java sendo utilizadas para proporcionar a geração de uma aplicação Java assinada e autenticada por certificados.

Como pode ser observado pela figura 5.8, uma aplicação Java deve, primeiramente, ser embutida em um arquivo JAR. O par de chaves e o certificado são gerados com a utilização de *keytool* e armazenados em um *keystore*. Tendo as chaves e o JAR, utiliza-se *jarsigner* para assinar a aplicação. Além disso, o certificado da chave pública pode ser exportado para qualquer entidade que desejar executar a aplicação. A geração de chaves, assinaturas e certificados a serem exportados são feitas através de comandos simples, explicitados na seção 6.3. Pode-se concluir, portanto, que as ferramentas *keytool* e *jarsigner*, quando utilizadas em conjunto, fornecem uma maneira simples e eficiente de tornar as aplicações Java confiáveis.

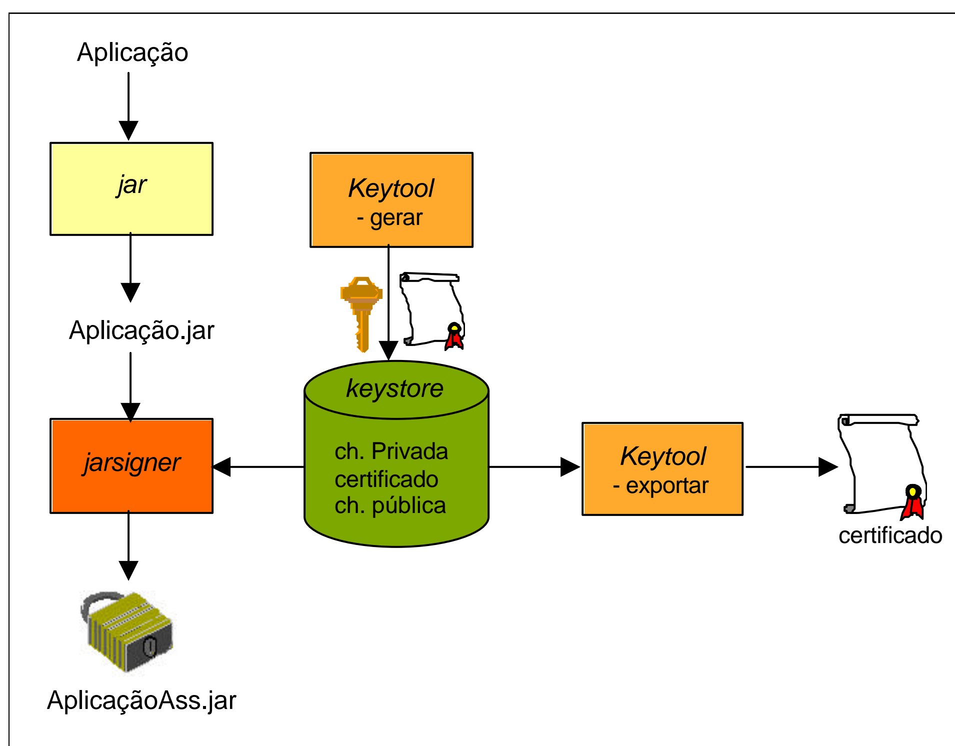

Figura 5.8 - Processo para assinatura de aplicações

A figura 5.9 ilustra o processo de autenticação de uma aplicação JAVA e sua execução de forma segura. Para isso, o certificado deve ter sido importado e aceito como válido. As permissões referentes ao assinante da aplicação devem estar definidas nos *policy files*, os quais podem ser gerados facilmente pela ferramenta *policytool*. Ao executar a aplicação, as permissões nos *policy files* serão concedidas através da verificação do certificado já importado para o *keystore*.

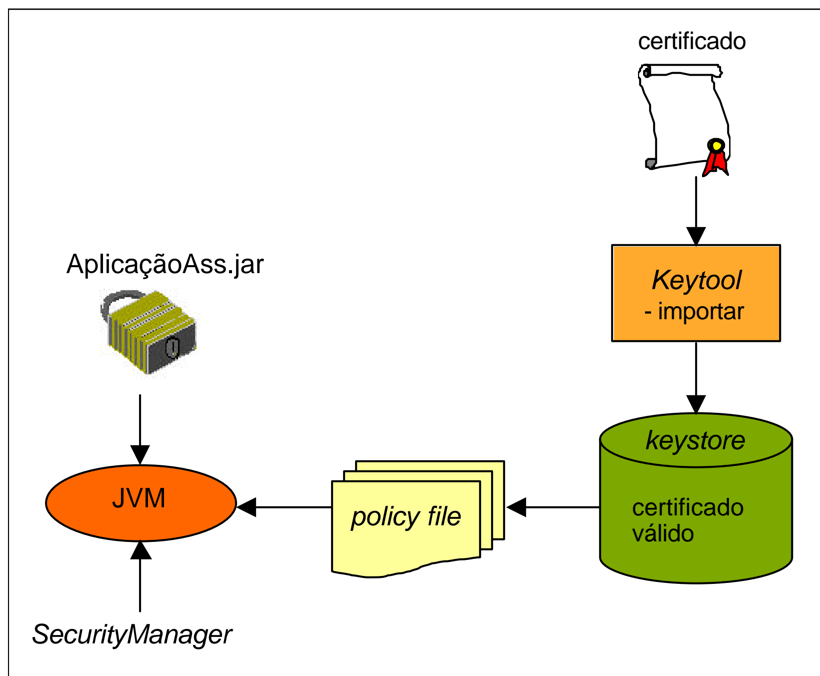

Figura 5.9- Processo de autenticação e execução de aplicações

As aplicações Java que desejarem utilizar a IBSU deverão estar devidamente assinadas e seu certificado deve ter sido importado. Para isso, elas devem utilizar-se das ferramentas *jar*, *keytool* e *jarsigner*.

O agente *Gateway* da IBSU irá verificar as assinaturas e os certificados através da API de segurança e das ferramentas. Se um agente for autenticado pelo *Gateway*, ele irá ser executado pelo *Pool*, que conta com uma política de segurança definida através de *policy files*. O capítulo seguinte apresenta, com detalhes, a implementação da IBSU.

Capítulo 6

Implementação da Interface

6.1 Considerações Iniciais

O capítulo 1 apresenta uma visão geral da Interface Básica para um Servidor Universal (IBSU) desenvolvida. Como ilustrado na figura 1.2, a IBSU deve estar posicionada entre o banco de dados, que faz parte do servidor universal, e as aplicações que desejarem acessar os objetos que ele armazena. Esse acesso é feito através de agentes de *software* que, de forma autônoma, irão acessar o banco de dados através dos métodos definidos em uma interface.

Este capítulo tem por objetivo descrever com maiores detalhes cada um dos módulos da IBSU: o *Gateway*, o *SecurityManager*, o *Pool* de Agentes e a Interface com o Banco de Dados.

O detalhamento dos módulos consiste na descrição da funcionalidade e dos recursos que foram utilizados para a implementação de cada um deles, considerando-se o embasamento teórico dos capítulos anteriores. Além disso, descreve-se também o perfil que os agentes que entrarão em contato com a IBSU devem possuir. Para finalizar é apresentado o InfoAgent, uma aplicação agente para demonstrar a funcionalidade da IBSU.

6.2 O Agente *Gateway*

O *Gateway* é um agente que faz parte da IBSU. Como ilustrado na figura 1.2, os agentes só poderão entrar no *Pool* de Agentes se passarem pelo *Gateway*. Ele é devidamente assinado e, como todos os outros agentes, será controlado pelo *Pool*. Sua função é promover a entrada de outros agentes no *Pool*, verificar a assinatura digital desses agentes e se o certificado do possuidor do agente é confiável.

O *Gateway* é independente dos demais agentes do *Pool*. Isso significa que se ele for retirado do *Pool*, os outros agentes irão continuar sua execução, mas novos agentes não poderão mais ter acesso ao *Pool*. Para cumprir a sua tarefa, o agente *Gateway* possui quatro classes principais, como ilustra a figura 6.1.

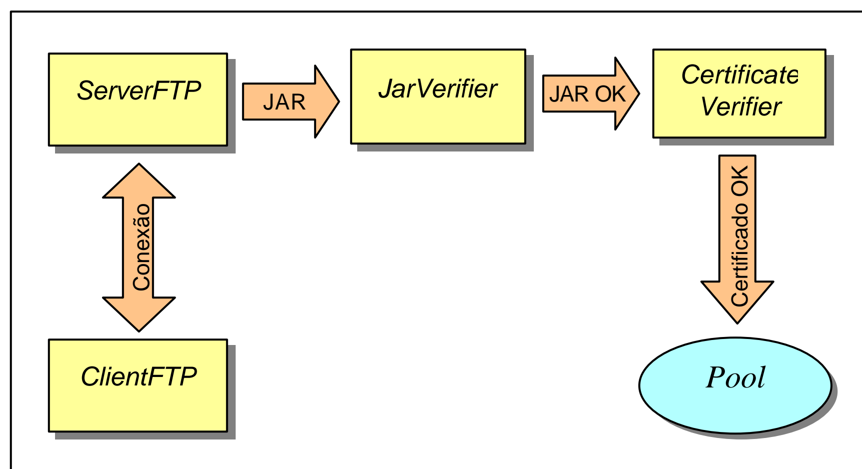

Figura 6.1 - O *Gateway*

É através da classe *ServerFPT* que os agentes irão entrar em contato com o *Gateway*. A função dessa classe é transferir o agente que está no cliente para o sistema de arquivos local. Para isso, é criado um *socket* que espera conexões em uma porta TCP.

A classe *ClientFTP* irá conectar-se com *ServerFTP* para enviar arquivos de extensão .jar contendo um agente. A classe *ClientFTP* oferece uma maneira simples e conveniente para que os usuários possam enviar seus agentes. Para estabelecer a conexão necessária para enviar um agente, o usuário deve chamar *ClientFTP* e passar o nome da máquina onde *ServerFTP* estiver executando seguido do número da porta. Após isso, o usuário deve digitar o nome do

arquivo JAR que contém o agente a ser enviado e o *ServerFTP* irá fazer uma cópia dele em seu sistema de arquivos local. A existência de *ClientFTP* não impede que os agentes sejam enviados de outra maneira. Pode-se desenvolver outras aplicações que se comuniquem diretamente com *ServerFTP*.

Tendo recebido o agente, o controle passa para a classe *JarVerifier*. Sua função é verificar se o agente está devidamente assinado. Este módulo garante que o conteúdo do agente não foi modificado durante a transferência. A verificação é feita através da geração de novos *hashes* e da comparação deles com os *hashes* antigos, como especifica a seção 5.4.2.3. Caso a assinatura seja válida, resta apenas verificar o certificado do assinante.

A classe *CertificateVerifier* tem como uma de suas funções extrair o certificado da chave pública do agente, a qual corresponde à chave privada que o assinou. Tendo esse certificado, será extraído o *alias* dele, ou seja, o nome de quem o assinou. Depois disso, o *CertificateVerifier* irá verificar se existe algum *alias*, igual ao extraído do certificado do agente, no *keystore* da IBSU. Caso exista, o certificado do agente é comparado ao certificado armazenado no *keystore* para aquele *alias*. Se os dois forem iguais, significa que o certificado daquele assinante já foi importado para o *keystore* da IBSU e aceito como válido. O *keystore* da IBSU chama-se `usstore` e fica armazenado em um arquivo.

Para que o certificado de um usuário da IBSU seja considerado válido, o grupo ao qual aquele usuário pertence já deve ter sido criado utilizando um método da Interface com o Banco de Dados, como melhor descrito na seção 6.6. Esses grupos devem ser associados a *aliases* no *keystore* da IBSU. No protótipo da IBSU existem dois grupos criados: *superusers* e *commonusers*.

Se a classe *CertificateVerifier* obtiver sucesso em sua verificação, significa que o certificado do assinante do agente é válido e, por isso, confiável. O *Gateway* pode então avisar ao *Pool* que há um agente pronto para entrar em execução. Para isso, o *Gateway* chama um método no *Pool*, específico para enviar os dados do agente, como é melhor descrito na seção 6.5. A partir do momento em que o agente começa a sua execução no *Pool*, a função do *Gateway* termina. Ele ficará novamente a espera de uma conexão.

É importante observar que, para um agente ser reconhecido e verificado com sucesso pelo *Gateway*, ele deve satisfazer alguns requisitos. A seção seguinte descreve o perfil dos agentes que entrarão em contato com o *Gateway*.

6.3 Os Agentes Usuários da IBSU

Como ilustrado na figura 1.2, as aplicações que entrarão em contato com o *Gateway* serão agentes de *software*. Através do *Gateway*, esses agentes poderão ganhar o direito de entrar em execução no *Pool* de Agentes ou não. Os agentes que serão enviados até o *Gateway* devem satisfazer, basicamente, aos seguintes requisitos para que possam entrar no *Pool*:

1. Serem desenvolvidos em Java.

2. Suas classes devem ser embutidas em um arquivo JAR (seção 5.4.2.1). Esse JAR deve conter um arquivo *manifest* que especifica qual é a classe que possui o método `main()` (Main-Class), necessário para iniciar a execução do JAR. Para gerar manualmente um JAR especificando uma Main-Class, basta criar um arquivo contendo o nome da Main-Class e digitar o seguinte comando:

```
jar cmf <arq_main> <nome_jar> <classes.class ou pacotes>
```

   `arq_main` é o nome do arquivo contendo o nome da Main-Class, `nome_jar` é o nome do JAR que será gerado, `classes.class ou pacotes` são as classes ou os pacotes que compõem a aplicação agente e que serão embutidos no JAR. Mais informações a respeito da ferramenta *jar* e de como modificar o *manifest* podem ser encontradas na *Sun Microsystems* [42].

3. Estarem devidamente assinados. A assinatura para um JAR, pode ser gerada através do seguinte comando:

```
jarsigner –keystore <nome_keystore> <nome_jar>
                <nome_alias>
```

   O comando irá pedir as senhas para o *keystore* e para a chave privada, e gerar o JAR assinado contendo o certificado da chave pública. É importante observar que, para gerar uma assinatura, já deve existir um *keystore* contendo a chave privada e o certificado da chave pública. Tendo gerado a assinatura, basta exportar o certificado e importá-lo para o *keystore* da IBSU, caso ele ainda não esteja lá. Mais informações a respeito das ferramentas para assinatura, dos passos para assinar um JAR e de como importar um certificado podem ser encontradas na *SunMicrosystems* [42] e em Dageforde [44].

4. A entidade que o assinou deve fazer parte de um grupo válido armazenado no banco de dados. A criação de grupos é descrita na seção 6.6.1.

6.4 O *SecurityManager*

Quando um agente entra em execução no *Pool*, ele pode ter acesso a todos os recursos que a plataforma Java pode oferecer. Ele pode, por exemplo, escrever em todo o sistema de arquivos local, se conectar a qualquer *host*, fazer introspeção (seção 3.4.3) em outras classes, etc. Mas, não é seguro que todos os agentes do *Pool* tenham liberdade irrestrita quanto aos recursos da plataforma Java. Um agente poderia modificar a política de segurança ou até mesmo atacar e danificar o sistema através de algum método de ataque descrito na seção 4.2. É para evitar ações que possam comprometer a segurança do sistema que existe o *SecurityManager*.

SecurityManager é uma classe Java que permite a implantação de uma política de segurança. Como descrito na seção 5.5, para definir uma política de segurança deve-se conceder permissões em um ou mais *policy files* ou então implementar uma subclasse do *SecurityManager*, o que é mais trabalhoso e suscetível a erros.

Esse projeto utilizou um *policy file* para promover a segurança. O *policy file* do protótipo da IBSU possui duas entradas, uma associada ao grupo *superusers* e a outra associada ao grupo *commonusers*. Para os *superusers* foram dadas todas as permissões possíveis (`AllPermission`). Já os *commonusers* possuem permissões para ler e escrever em alguns arquivos e estabelecer conexões com alguns *hosts*.

A partir do momento em que o *SecurityManager* for inicializado, as permissões concedidas nos *policy files* passam a vigorar. Então, se um agente deseja, por exemplo, deletar um arquivo local, ele pode chamar o método `java.io.File.delete()`. Como esse método não é seguro, haverá uma chamada automática para o método `checkDelete(String)` na classe *SecurityManager*, que irá verificar nos *policy files* se o agente que está tentando deletar o arquivo possui permissão para isso. Caso o agente pertença ao grupo *superuser*, ou a qualquer outro grupo que possua permissão nos *policy files* para apagar arquivos, o *SecurityManager* deixará que o arquivo seja apagado. Caso contrário uma *Security Exception* será gerada.

O nome dos *policy files* que irão definir a política de segurança do sistema devem ser explicitados em um arquivo especial, que define as propriedades de segurança de toda a plataforma Java. Este arquivo se chama `java.security` e se encontra no subdiretório `lib/security` do diretório *home* do ambiente Java (JRE)..

Com a utilização dos *policy files*, torna-se simples a adição de mais grupos usuários da IBSU, além dos *superusers* e *commonusers*. Basta criar um grupo, como descreve a seção 6.6, e conceder permissões associadas a ele no *policy file*, o que pode ser feito de forma simples através da ferramenta *policytool*. A mesma facilidade não seria alcançada caso o *SecurityManager* fosse uma subclasse, pois sempre que um novo grupo fosse criado, com mais ou menos prioridades que os já existentes, dever-se-ia modificar o código do *SecurityManager*, o que só pode ser feito por um programador experiente.

6.5 O *Pool* de Agentes

O *Pool* de Agentes é um ambiente onde todos os agentes irão estar executando. Ele é responsável por colocar os agentes em execução e controlar o tempo de vida de cada um deles. Para realizar essas tarefas, o *Pool* de Agentes é composto de basicamente três classes principais: *Pool*, *JarRunner* e *TimeCounter*, como ilustrado na figura 6.2.

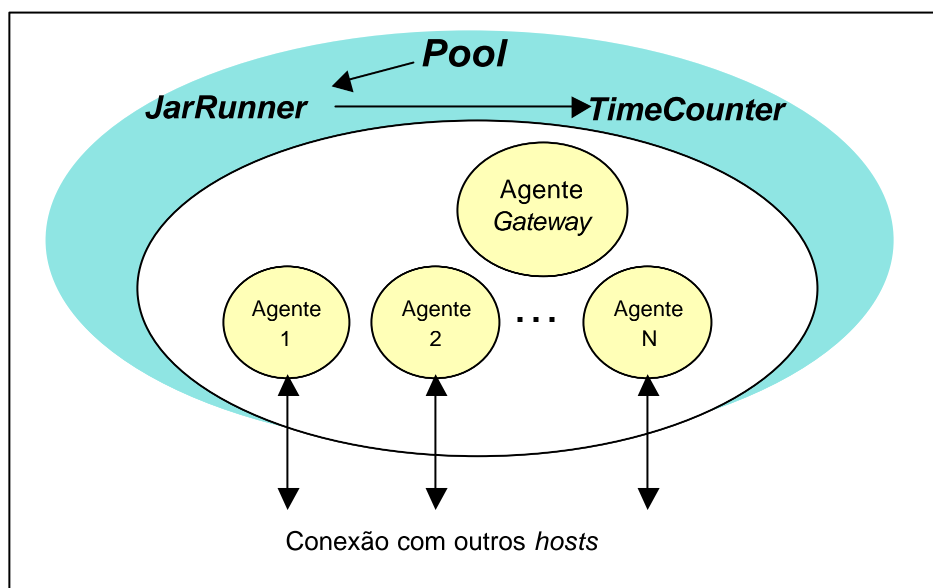

Figura 6.2 - O *Pool* de Agentes

A classe *Pool* deve ser a primeira a ser chamada. Quando ela é executada, a primeira ação tomada é colocar o agente *Gateway* em execução. Assim, mais agentes podem passar por ele e, caso sejam verificados com sucesso, entrar no *Pool* de Agentes.

A execução do *Gateway* e de qualquer outro agente é feita pela classe *JarRunner*, através do disparo de um *thread*. É a *JarRunner* que instala o *SecurityManager*. Os parâmetros para ela são o grupo ao qual pertence o dono do agente e o *path* do agente. O *path* é necessário para que o agente possa ser executado e suas classes carregadas. O grupo ao qual pertence o agente será necessário para o controle de acesso aos métodos da Interface com o Banco de Dados.

A classe *JarRunner* irá começar a execução de um agente logo depois que o *Gateway* terminar sua verificação. Para isso, a classe *Pool* possui o método `receiveJarInfo(String alias, String path)` que é chamado pelo *Gateway* para mandar o grupo de um agente e o seu *path*. Quando as informações chegam, o classe *Pool* chama *JarRunner*, que dispara em *thread*, iniciando a execução de mais um agente.

A classe *TimeCounter* irá controlar o tempo de vida de cada agente. Ela é chamada por *JarRunner* quando um agente entra em execução. Para que o controle do tempo possa ser feito, *TimeCounter* dispara um *thread* no momento em que o agente entra em execução. Esse *thread* é colocado em estado dormente (*sleep state*) durante o tempo correspondente ao tempo de vida concedido ao agente. Depois que esse tempo acaba, o *thread* é acordado e, caso o *thread* agente ainda esteja em execução, ele o mata. O tempo de vida de um agente pode variar de segundos a anos.

6.6 A Interface com o Banco de Dados

É no banco de dados onde está armazenada a informação em *roots* (pontos de entrada para objetos). A Interface com o Banco de Dados consiste na declaração dos métodos que irão possibilitar aos agentes manipularem os *roots*, os grupos, fazer associações entre eles para leitura e/ou escrita e obter informações sobre eles. A Interface com o Banco de Dados, como qualquer outra interface Java, não possui a implementação de nenhum método. Ela apenas define o cabeçalho dos métodos, especificando os parâmetros que devem ser passados e o tipo que deve ser retornado para cada método.

Como descrito na seção 2.2, para armazenar objetos em um servidor universal, pode-se utilizar dois tipos de SGBD: os SGBDOOs, como *ObjectStore*, *ObjectStore PSE* e *Jasmine* ou então um SGBDOR, como *Oracle*, *Informix* e *DB2*. Para utilizar a IBSU com um desses SGBDs, bastaria criar uma classe Java que implementasse os métodos definidos na Interface com o Banco de Dados para o SGBD escolhido.

Os métodos definidos na Interface com o Banco de Dados podem ser classificados em três grupos: para manipulação de grupos de usuários, para manipulação de *roots* e para controle dos *threads* de execução dos agentes. Toda a informação que a IBSU necessita para executar é armazenada no próprio SGBD com o qual ela faz a interface, com exceção do *keystore* que o ambiente Java exige que fique em um arquivo. Isso garante que a IBSU seja tão robusta quanto o SGBD para o qual ela faz interface.

As definições dos métodos, assim como a descrição da funcionalidade de cada um, são dadas nas subseções 6.6.1 a 6.6.3 a seguir.

6.6.1 Manipulação de Grupos

A Interface com o Banco de Dados define os seguintes métodos para a manipulação de grupos:

- `public void initialize()`: Esse método inicializa o banco de dados com o(s) primeiro(s) grupo(s). Grupos são necessários para que os agentes possam ser executados. O grupo de quem assinou cada agente já deve estar cadastrado no banco de dados quando o agente chegar.

- `public void createGroup(String groupName, long time)`: Cria um novo grupo no banco de dados, especificando o tempo de vida que os agentes pertencentes ao grupo terão.

- `public void deleteGroup(String groupName)`: Deleta um grupo existente no banco de dados.

- `public String[] getRootsToRead(String groupName)`: Retorna um *array* de `String` contendo o nome dos *roots* que o grupo passado como parâmetro possui permissão para leitura.

- `public String[] getRootsToWrite(String groupName)`: Retorna um *array* de `String` contendo o nome dos *roots* que o grupo passado como parâmetro possui permissão para atualizar e deletar.
- `public long getTime(String groupName)`: Retorna o tempo que os agentes de determinado grupo podem ficar em execução.
- `public boolean isGroup(String groupName)`: Retorna `true` caso já exista no banco de dados um grupo com o nome passado como parâmetro.
- `public String[] getGroups()`: Retorna um *array* de `String` contendo o nome de todos os grupos armazenados no banco de dados.

6.6.2 Manipulação de *Roots*

A Interface com o Banco de Dados define os seguintes métodos para a manipulação de *roots*:

- `public void createRoot(String rootName)`: Cria um *root* vazio no banco de dados.
- `public Object get(String rootName)`: Retorna um `Object` contendo o conteúdo do *root* solicitado. `Object` é a superclasse de todas as classes Java. Isso significa que o tipo de um *root* pode ser qualquer classe Java.
- `public void deleteRoot(String rootName)`: Apaga um grupo existente no banco de dados.
- `public void set(String rootName, Object root)`: Atualiza os dados de um *root*. Os dados a serem atualizados devem estar armazenados em um `Object`.
- `public void setGroupToRead(String rootName, String groupToRead)`: Dá permissão para que um grupo possa ler determinado *root*.

- `public void setGroupToWrite(String rootName, String groupToWrite)`: Dá permissão para que um grupo possa atualizar e apagar determinado *root*.

- `public String[] getRoots()`: Retorna um *array* de `String` contendo o nome de todos os *roots* armazenados no banco de dados.

- `public boolean isRoot(String rootName)`: Retorna `true` caso o *root* em questão já exista no banco de dados.

- `public String[] getGroupsToRead(String rootName)`: Retorna um *array* de `String` contendo o nome de todos os grupos que possuem permissão de leitura para o *root* em questão.

- `public String[] getGroupsToWrite(String rootName)`: Retorna um *array* de `String` contendo o nome de todos os grupos que possuem permissão de escrita e deleção para o *root* em questão.

- `public boolean isReadable(String rootName, String groupName)`: Retorna `true`, caso um grupo tenha permissão de leitura para determinado *root*.

- `public boolean isWriteable(String rootName, String groupName)`: Retorna `true`, caso um grupo tenha permissão de escrita e deleção para determinado *root*.

6.6.3 Controle de *Threads*

A Interface com o Banco de Dados define os seguintes métodos para o controle dos agentes e seus respectivos *threads* de execução:

- `public void associate (String threadName, String groupName, String agentName)`: Associa um *thread* de execução ao grupo do agente e ao nome do agente que está sendo executado neste *thread*. Este método é chamado na classe *JarRunner* (pertencente ao *Pool* de Agentes) sempre que um novo agente entrar em execução. Como cada agente deve ser executado

por um *thread*, é possível controlar a que grupo pertence cada *thread*. Esse controle é necessário para estabelecer o controle de acesso aos métodos da Interface com o Banco de Dados. Dependendo do grupo ao qual pertence o *thread*, o acesso a um método será permitido ou não.

- `public String getGroup (String threadName)`: Retorna uma `String` contendo o nome do grupo associado ao *thread* em questão.
- `public String getAgentName (String threadName)`: Retorna uma `String` contendo o nome do agente associado ao *thread* em questão.
- `public void removeAssociation(String threadName)`: Desfaz a associação de um agente com um *thread*. Deve ser chamado quando terminar a execução ou o tempo de vida do agente.

6.6.4 Implementação de um Banco de Dados

Um dos objetivos deste projeto é a definição de uma interface para um banco de dados que possa fazer parte de um servidor universal. Tal interface pode ser implementada para qualquer banco de dados. Como o tipo do banco de dados não possui tanta relevância para este projeto, foi implementado para testes um banco de dados simples, que armazena os dados em memória.

O ambiente Java possibilita o armazenamento e recuperação de objetos através de um recurso chamado Serialização (seção 3.4.2). Serializar um objeto Java consiste em salvá-lo na forma de um *stream*, ou seja, uma seqüência de *bytes* que descreve totalmente o objeto, o que possibilita a recuperação do objeto em sua forma original. A utilização de *streams* ao invés de um SGBD real, para armazenamento não transitório de objetos, apresenta várias limitações, entre elas:

- O arquivo serializado contendo os *roots* deve ser lido inteiramente para a memória, o que é um problema caso o arquivo seja muito grande.
- Se algum problema acontecer durante a serialização e escrita de um *root* em disco, o arquivo serializado pode ser corrompido e as informações do *root* perdidas.

Para armazenar os dados, o que inclui os *roots*, os grupos, a associação de permissões e informações sobre os *threads* em execução, foram criadas cinco tabelas *hash*, as quais mapeam chaves a valores. A biblioteca básica de Java possui uma classe chamada `Hashtable` que contém métodos para manipular as chaves e os valores mapeados por elas. As chaves e seus respectivos valores podem ser qualquer classe Java. As cinco tabelas *hash*, que compõem o banco de dados, são descritas a seguir:

- **Tabela `rootsDataBase`**: Essa tabela relaciona o nome de cada *root* ao seu conteúdo. A chave é o nome do *root*, que é do tipo `String` e deve ser especificada na criação do *root*. As chaves mapeam valores do tipo `Object`, que irão armazenar o conteúdo dos *roots*. A `Hashtable` é serializada para o arquivo `roots.out`.

- **Tabela `groupsDataBase`**: Essa tabela relaciona cada grupo com seu devido tempo de execução. Quando um grupo é criado ele recebe um nome, do tipo `String`, e um tempo, do tipo `long`. Esses dados são inseridos em uma `Hashtable`, que é serializada para o arquivo `groups.out`. A chave é o nome do grupo e o valor mapeado é o tempo, que deve ser estabelecido em milisegundos.

- **Tabela `toReadDataBase`**: Essa tabela relaciona cada *root* com os grupos que possuem permissão para lê-lo. A chave é o nome do *root*, do tipo `String`, e os valores são mapeados para um *array* de `String` contendo o nome dos grupos que possuem permissão para ler o *root* especificado. A `Hashtable` é serializada para um arquivo chamado `toRead.out`.

- **Tabela `toWriteDataBase`**: Essa tabela é como a `toReadDataBase`, mas armazena os grupos que possuem permissão para escrever no *root*, o que inclui deleção. A `Hashtable` é serializada para o arquivo `toWrite.out`.

- **Tabela `threadsDataBase`**: Essa tabela relaciona cada *thread* em execução com o nome do agente, que está sendo executado por aquele *thread*, e com o nome do grupo ao qual o agente pertence. A chave para cada entrada na tabela *hash* é o nome do *thread*, do tipo `String`, e os valores são mapeados para um *array* de

`String`, contendo o nome do agente e o nome do grupo ao qual o agente pertence. A `Hashtable` é serializada para o arquivo `threads.out`.

A figura 6.3 ilustra o banco de dados implementado, onde os agentes poderão manipular grupos, *roots* e estabelecer permissões de forma transparente, ou seja, sem ter o conhecimento de que os dados manipulados estão armazenados em *streams*.

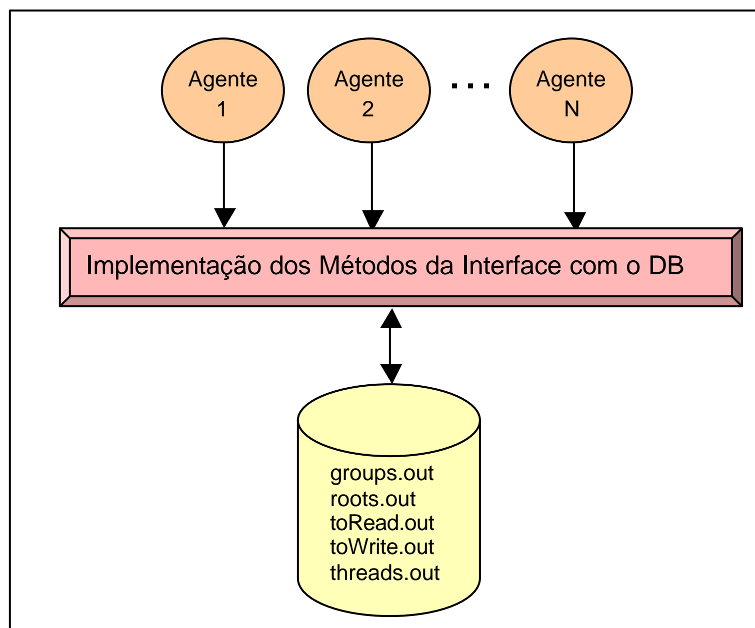

Figura 6.3 – Arquitetura de Acesso ao Banco de Dados

Os métodos da Interface com o Banco de Dados foram implementados na classe `DBImplementation`. Cada método foi desenvolvido seguindo as especificações da Interface.

6.7 O InfoAgent

O InfoAgent é um agente exemplo que foi desenvolvido para ser executado pela IBSU. Sua função é dar informações a respeito do estado da IBSU em um determinado instante. Ele abre uma porta TCP e fica a espera de conexões vindas a partir de um *browser* (Internet *Explorer* ou *Netscape Communicator*). Ao receber uma requisição, ele envia uma página HTML contendo informações sobre a IBSU, tais como: o número de grupos cadastrados, os agentes que estão executando e a que grupo pertence cada agente em

execução. Para estabelecer uma conexão com o InfoAgent basta utilizar um *web browser* e acessar o endereço da máquina onde a IBSU com o InfoAgent estiver instalada. A resposta aos clientes, caso o InfoAgent estivesse executando na máquina `terere.intermidia.icmc.sc.usp.br`, seria parecida com a figura 6.4.

Figura 6.4 - Exemplo de página enviada pelo InfoAgent

Para ser executado pelo *Pool*, o InfoAgent satisfaz a todos os requisitos descritos na seção 6.3. Sua assinatura pertence ao grupo de *superusers*, o qual já está cadastrado no banco de dados e possui um certificado confiável no *keystore* da IBSU.

O InfoAgent serve como um demonstrativo do funcionamento da IBSU, pois para responder às requisições dos clientes, ele deve ser autenticado pelo *Gateway* e entrar em execução no *Pool* de Agentes. Ele também serve como ferramenta de *debug* da IBSU, pois ele irá fornecer informações a respeito do estado da IBSU (grupos cadastrados e agentes em execução) em determinado instante.

6.8 Considerações Finais

Este capítulo descreve a maneira como cada módulo da IBSU foi implementado, os requisitos que um agente deve satisfazer para ser executado pelo *Pool* e apresenta o InfoAgent, um agente exemplo que demonstra a funcionalidade da IBSU e fornece informações a respeito do estado interno da IBSU em determinado instante.

As classes e métodos comentados neste capítulo são os mais relevantes para a IBSU. Um detalhamento maior, que englobe todas as classes utilizadas no desenvolvimento da IBSU, com seus respectivos métodos, e um tutorial para o uso da IBSU, podem ser encontrados em Linhalis[47].

Capítulo 7

Conclusões

Neste capítulo, primeiramente, é apresentada uma visão geral do projeto, expondo seu objetivo inicial. Em seguida, são expostos os resultados obtidos e as contribuições que eles acarretam. Para finalizar, são feitas algumas sugestões para futuros trabalhos.

7.1 Visão Geral

Este projeto desenvolveu uma Interface Básica para um Servidor Universal (IBSU) que deve estar posicionada entre o banco de dados de um servidor universal e os agentes que desejam acessar e/ou armazenar objetos nesse banco de dados.

A IBSU fornece um ambiente de execução aberto e ao mesmo tempo seguro para os agentes. Assim, os agentes podem utilizar os recursos do sistema para se conectarem a recursos externos, o que promove a abertura do ambiente de execução. Os agentes podem também armazenar e recuperar *roots* do banco de dados. Tanto o acesso a recursos do sistema quanto aos *roots* do banco de dados devem ser feitos de forma segura, ou seja, de acordo com as permissões associadas ao grupo ao qual o agente pertence.

7.2 Resultados e Contribuições

A IBSU foi desenvolvida em Java e possibilita que vários agentes possam estar executando, de forma concorrente, em um *Pool* de Agentes. O *Pool* é o ambiente de execução dos agentes. Os agentes que estiverem no *Pool* devem executar de forma segura com relação ao acesso aos recursos do sistema e aos *roots* do banco de dados. Para isso, três providências são tomadas:

- Verificação da autenticidade do agente, que é promovida pelo agente *Gateway* através de assinaturas digitais e certificados. Os agentes só poderão entrar no *Pool* caso sejam verificados com sucesso pelo *Gateway*.
- Garantia da segurança dos recursos do sistema, o que é feito pelo *SecurityManager*. Ele irá verificar as permissões associadas ao grupo dos agentes sempre que eles tentarem executar uma operação que pode comprometer a segurança do sistema. A verificação é feita de forma automática, pois o *SecurityManager* é uma classe Java que, se instalada, consulta os arquivos de definição da política do sistema (*policy files*), onde as permissões referentes a cada grupo devem estar explícitas.
- Controle de acesso aos *roots* do banco de dados, o que é garantido pela Interface com o Banco de Dados. Os agentes podem realizar consultas somente através dos métodos definidos pela Interface com o Banco de Dados. Ela define métodos para manipulação de grupos, *roots* e controle de permissões para acesso aos *roots*.

Conclui-se, então, que com a IBSU é possível que agentes executem de forma aberta e acessem objetos armazenados em bancos de dados de servidores universais de forma segura. Além disso, os agentes utilizarão Java como linguagem de consulta, que é mais completa e flexível do que as linguagens utilizadas exclusivamente para esse fim.

7.3 Trabalhos futuros

A atual implementação da IBSU está em sua primeira versão, ela pode ser melhorada em muitos pontos. A Interface com o Banco de Dados é apenas um protótipo que serviu, basicamente, para a relização de testes.

A Interface com o Banco de Dados define os métodos básicos para a manipulação de objetos no banco de dados de um servidor universal. Assim, pode-se implementar seus métodos para acessar objetos armazenados em SGBDOOs ou SGBDORs, dependendo das necessidades de cada aplicação. Seria conveniente desenvolver uma biblioteca de classes que implementassem essa interface para vários SGBDs comerciais, como o *Informix*, *Oracle*, *PSE*, *Jasmine*, etc.

Outra opção de desenvolvimento seria implementar um *Security Manager* como uma subclasse da classe *SecurityManager* do ambiente Java, ao invés de implementá-lo utilizando *policy files*. Dessa forma, seria possível pesquisar novos tipos de domínios de proteção. Esses novos tipos poderiam utilizar informações dinâmicas a respeito da origem dos agentes, o que não pode ser definido nos *policy files* por eles serem estáticos.

Uma linha muito interessante de pesquisa seria criar uma classe que implementasse a Interface do Banco de Dados com JavaSpaces [48]. O JavaSpaces é uma tecnologia que permite o armazenamento persistente de objetos Java que podem ser compartilhados em rede (utilizando a tecnologia Jini). Mais que um banco de dados, o JavaSpace permite a sincronização de vários programas numa rede, sendo ideal para aplicações que utilizam *clusters* de computadores ligados por uma rede de alta velocidade. Fazendo a interface entre os agentes e os serviços e dados de um JavaSpace a IBSU seria capaz não só de servir dados, mas capacidade de processamento, acesso a periféricos e outros serviços diversos, implementando assim um servidor realmente universal.

Referências Bibliográficas

1. COULOURIS G., DOLLIMORE, J., KINDBERG, T. **Distributed systems**: concepts and design. 2. ed. Great Britain: Addison-Wesley, 1994. 644p.

2. TANENBAUM, A. S. **Distributed operating system**. New Jersey: Prentice-Hall, 1995. 614p.

3. ROGERS, P. **Object database mangement systems**. 1997. [online]. [30/04/99]. Disponível na Internet: <http://www.odi.com>

4. OBJECT DESIGN. **Choosing an object database**. Burlington: 1997. [online]. [30/04/99]. Disponível na Internet: <http://www.odi.com>

5. ABEERDEN GROUP. **Universal servers**: RDBMS technology for the next decade. Boston: 1995. [online]. [22/04/00]. Disponível na Intenet: <http://www.informix.com/whitepapers/aberdeen/aberdeen.htm>

6. DAVIS, J.R. Universal servers: the players, part2. **DBMS**, v. 10, n. 8, p. 75-81, 1997. [online]. [22/04/00]. Disponível na Internet: <http://www.dbmsmag.com/9707d14.html>

7. ORACLE CORPORATION. **Oracle security server guide release 2.0.3**. 1997. [online]. [22/04/00]. Disponível na Internet: <http://shelob.nevalink.ru/oracle/doc/network803/A54088_01/toc.htm>

8. CHOKHANI, S., FORD, W. **Internet X.509 public key infrastructure certificate policy and certification practices framework**. Mar. 1999. [online]. [21/05/2000]. Disponível na Internet: <http://www.imc.org/rfc2527>

9. INFORMIX SOFTWARE. **INFORMIX-Universal server trusted facility manual**. [online]. [14/12/99]. Disponível na Internet: <http://lighwww.epfl.ch/docs/informix/answers/answers/pubs/pdf4808.pdf>

10. MOREIRA, D. A., WALCZOWSKI, L. T. Using software agents to generate VLSI layouts. **IEEE Expert Intelligent Systems**, v. 12, n. 6, p. 26-32, 1997.

11. MAES, P. Agents that reduce work and information overload. **Communications of the**

ACM, v. 37, n. 7, p. 31-40, 1994.

12. MAES, P. Artificial life meets entertainment: lifelike autonomous agents. **Communications of the ACM**, v. 38, n. 11, p. 108-114, 1995.

13. SMITH, D. C., CYPHER, A., SPOHRER, J. KidSim: programming agents without a programming language. **Communications of the ACM**, v. 37, n. 7, p. 55-67, 1994.

14. FRANKLIN, S., GRAESSER, A. Is it na agent or just a program?: a taxonomy for automonous agents, In: INTERNATIONAL WORKSHOP ON AGENT THRORIES, ARCHITECTURE AND LANGUAGES, 3, 1996. [online]. [14/01/2000]. Disponível na Internet: <http://www.msci.memphis.edu/~franklin/AgentProg.html>

15. NETO, A.B., GOUVEIA, D., SILVA, M. J. ACE: um agente de compras na Internet, In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 12, Maringá**,** 1998. *Anais*. Maringá, UEM, 1998, p. 281-296.

16. VAN HOFF, A. Java: getting down to business. **Dr. Dobb's Journal**, n. 281, p. 20-24, 1998.

17. ECKEL, B. **Thinking in Java**. New Jersey: Prentice Hall, 1998. 859p. [online]. [22/04/00]. Disponível na Internet: <http://www.bruceeckel.com/javabook.html>

18. GOSLING, J. The feel of Java. **IEEE Computer**, v. 30, n. 6 p. 53-57, 1997.

19. COCKBURN, A. A. R. The impact of object-orientation on application development. **IBM Systems Journal**, v. 38, n. 2&3, p. 308-332, 1999.

20. RUMBAUGH, J., BLAHA, M., PREMERLANI, W., EDDY, F., LORENSEN, W. **Object-oriented modeling and design**. Englewood Cliffs: Prentice-Hall, 1991. 500p.

21. WONG, D., PACIOREK, N., MOORE, D. Java-based mobile agents. **Communications of the ACM**, v. 42, n. 3, p. 92-102, 1999.

22. HAMILTON, G. **JavaBeans API specification 1.01**, Jul. 1997. [online]. [22/04/00]. Disponível na Internet: <http://www.java.sun.com/beans/docs/spec.html>

23. SUN MICROSYSTEMS. **Java object serialization specification**. Nov. 1998. [online]. [22/04/00]. Disponível na Internet: <http://www.java.sun.com/products/jdk/1.2/download-pdf-ps.html>

24. SUN MICROSYSTEMS. **Java core reflection - API and specification**, Jan. 1997. [online]. [22/04/00]. Disponível na Internet: <http://www.java.sun.com/products/jdk/1.1/download-pdf-ps.html>

25. TREMBLETT, P. Java reflection. **Dr. Doob's Journal**, n. 281, p.36-39, 1998.

26. BERNARDES, Mauro César. **Avaliação do uso de agentes móveis em segurança computacional**. São Carlos, 1999. 112p. Dissertação (Mestrado) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo.

27. CHIN, S. K. High-confidence design for security. **Communications of the ACM**, v. 42, n. 37, p. 33-37, 1999.

28. FIPS PUB 46-2. **Data encryption standard**. Dez. 1993. [online]. [07/05/2000]. Disponível na Internet: <http://www.itl.nist.gov/div897/pubs/fip46-2.htm>

29. PKCS#1. **RSA cryptography standard**, versão 2.0. Set. 1998. [online]. [07/05/2000]. Disponível na Internet: <http://www.rsasecurity.com/rsalabs/pkcs/pkcs-1/index.html>

30. CLARK, A.J. Cryptographic controls - the eternal triangle. **Computers & Security**, v. 15, n. 7, p. 576-584, 1996.

31. BHIMANI, A. Securing the commercial internet. **Communications of the ACM**, v. 39, n. 6, p. 29-35, 1996.

32. FIPS PUB 180-1. **Secure hash standard**. Abr. 1995. [online]. [07/05/2000]. Disponível na Internet: <http://www.itl.nist.gov/div897/pubs/fip180-1.htm>

33. KALISKI, B. **The MD2 message-digest algorithm**, RFC 1319. Abr. 1992. [online]. [07/05/2000]. Disponível na Internet: <http://www.cis.ohio-state.edu/htbin/rfc/rfc1319.html>

34. RIVEST, R. **The MD5 message-digest algorithm**, RFC 1321. Abr. 1992. [online]. [07/05/2000]. Disponível na Internet: <http://www.cis.ohio-state.edu/htbin/rfc/rfc1321.html>

35. GONG, L. Java security: present and near future. **IEEE Micro**, v. 17, n. 03, p. 14-19, 1997.

36. ZUKOWSKI, J., ROHALY, T. **Fundamentals of Java security**. [online]. [26/02/2000]. Disponível na Internet: <http://developer.java.sun.com/developer/onlineTraining/Security/Fundamentals/abstract.html>

37. KOVED, L. et. al. The evolution of Java security. **IBM Systems Journal**, v. 37, n. 3, p. 349-364, 1998.

38. LINDHOLM, T.; YELLING, F. **The Java virtual machine specification**, 2. ed. [online]. [26/02/2000]. Disponível na Internet: <http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html>

39. MELOAN, S. **Fine-grained security - The key to network safety**: Strength in flexibility. Nov. 1999. [online]. [19/11/1999]. Disponível na Internet: <www.java.sun.com>

40. GONG, L. **Java security architecture**. Out. 1998. [online]. [17/03/200]. Disponível na Internet: <www.java.sun.com/products/jdk/1.3/docs/guide/security/spec/security-spec.doc.html>

41. SUN MICROSYSTEMS. **Java cryptography architecture API specification and reference**. Dec. 1999. [online]. [18/03/2000]. Disponível na Internet: <www.java.sun.com/products/jdk/1.2/docs/guide/security/CryptoSpec.html>

42. SUN MICROSYSTEMS. **Java 2 SDK tools**. [online]. [18/03/2000]. Disponível na Internet: <www.java.sun.com/products/jdk/1.2/docs/tooldocs/tools.html>

43. SOMMERER, A. **The Java tutorial**: trail - JAR files. [online]. [18/03/2000]. Disponível na Internet: <http://web2.java.sun.com/docs/books/tutorial/jar/index.html>

44. DAGEFORDE, M. **The Java tutorial**: trail - security in Java 2 SDK 1.2. [online]. [18/03/2000]. Disponível na Internet: <http://web2.java.sun.com/docs/books/tutorial/security1.2/index.html>

45. SUN MICROSYSTEMS. **Permissions in the Java 2 SDK**. Out. 1998. [online]. [18/03/2000]. Disponível na Internet: <http://www.java.sun.com/products/jdk/1.2/docs/guide/security/permissions.html>

46. SUN MICROSYSTEMS. **Default policy implementation and policy file syntax**. Out. 1998. [online]. [18/03/2000]. Disponível na Internet: <http://www.java.sun.com/products/jdk/1.2/docs/guide/security/PolicyFiles.html>

47. LINHALIS, F. **The basic interface for an universal server tutorial**. São Carlos: ICMC/USP, s.d. (Relatório Técnico). (no prelo).

48. SUN MICROSYSTEMS. **JavaSpaces technology**. [online]. [21/05/2000]. Disponível na Internet: <http://java.sun.com/products/javaspaces>