Interface Básica para um Servidor Universal

Flávia Linhalis
Dilvan de Abreu Moreira

Departamento de Ciências de Computação e Estatística
Instituto de Ciências Matemáticas e de Computação - ICMC - USP
Av. Dr. Carlos Botelho, 1465 - Caixa Postal 668 - 13560-970 São Carlos, SP, Brasil
{flavia, dilvan}@icmc.sc.usp.br

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 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 (entry points for objects) and systems resources. However, the BIUS 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 allow 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 Pool of Agents runs the agents and controls 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 not entitled to use.

The BIUS and the software agents that run in the Pool are implemented in Java. This agents have in an universal server the same role as query languages (such as SQL) have in a relational database server. 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.

Keywords: interface, universal server, authentication, access control, software agents.

O Conceito de Servidor Universal

Existem atualmente muitos tipos de programas servidores de informação disponíveis, tais como servidores de HTML (HyperText Markup Language), de SQL (Structured Query Language), 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 a manterem vários tipos de servidores executando e, muitas vezes, armazenando a mesma informação em vários formatos diferentes. Seria então extremamente útil ter toda a informação armazenada 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 modos. 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. Existe, então, apenas uma fonte de dados a ser mantida e muitas maneiras de acessá-la e modificá-la.

O banco de dados de um servidor universal deve ser capaz de armazenar qualquer tipo de objeto. Para isso, a indústria de bancos de dados oferece 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).

Como os dados de 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.

A Interface

A Interface Básica para um Servidor Universal (IBSU), descrita neste artigo, 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 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 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.

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 dá uma visão geral da IBSU, que é composta de quatro componentes: a Interface com o Banco de Dados, o Gateway, o Pool de Agentes e o SecurityManager.

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 lhe 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, verifica a assinatura digital e os certificados daquele agente. Caso a verificação seja feita com sucesso, significa que o dono daquele agente é confiável e, por isso, o agente 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.

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 oferece. 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 Security Manager 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 Interface 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.

Java e os Agentes de Software

Os agentes de software (Franklin e Graesser, 1996) (Maes, 1995) foram inventados para facilitar a criação

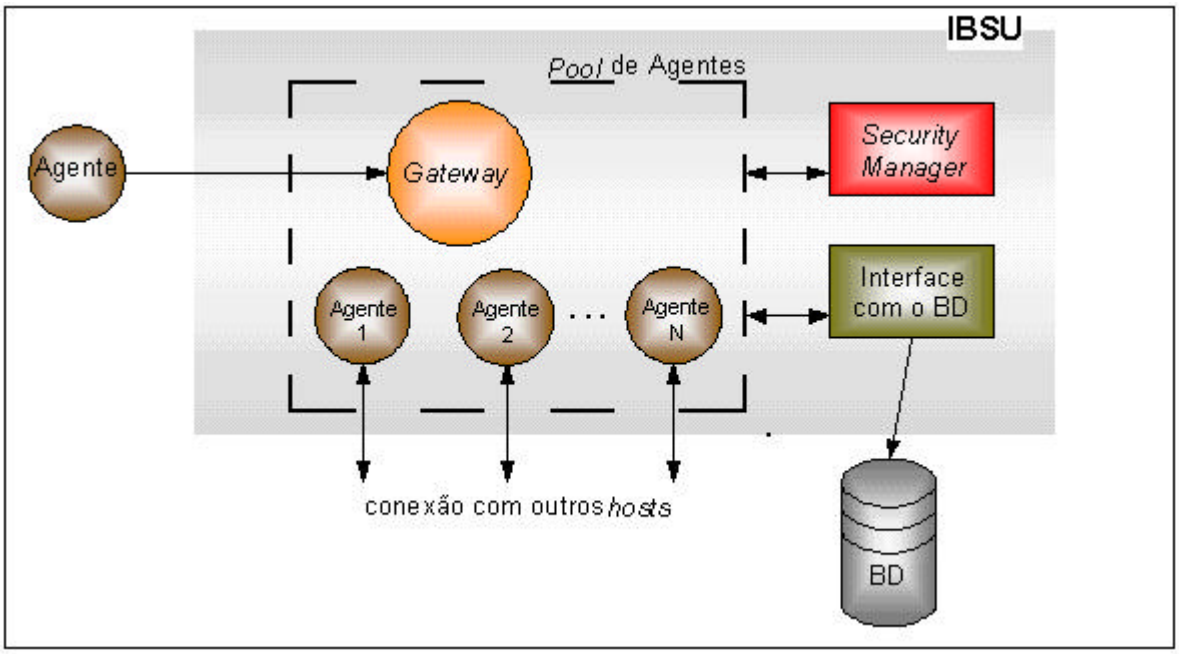

Figura 1 – Visão Geral da IBSU

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.

A IBSU e os agentes que entrarão em contato com ela são desenvolvidos em Java, pois 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 (Wong et.al 1999). Além da portabilidade, Java possui várias outras características adequadas para a implementação de agentes, entre elas:

Java Beans: 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. Um bean (Hamilton, 1997) é um componente que pode trabalhar em conjunto com outros beans. Não é necessário escrever nenhum código extra ou usar extensões especiais de Java para desenvolver um bean, pois um bean é simplesmente uma classe.

Serialização: A capacidade de armazenar e recuperar objetos Java é chamada Serialização (Sun, 1998a). 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. Entre as vantagens na serialização de objetos pode-se citar a portabilidade necessária para objetos remotos e o suporte à persistência. Nesse segundo caso, uma classe serializável pode, até mesmo, fazer o papel de banco de dados para uma aplicação.

Introspecção: Uma aplicação pode utilizar-se de introspecção (Eckel, 1998) (Sun, 1997) para detectar os métodos e variáveis disponíveis em uma classe e produzir seus nomes sem ter acesso ao código fonte. 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.

Class Loading: O class loader permite que as classes que compõem uma aplicação (agente ou não) possam ser carregadas em tempo de execução (Wong et al. 1999).

Segurança: A segurança de Java impõe restrições para as classes carregadas, impedindo que aplicações não confiáveis tenham liberdade irrestrita. A segurança para a execução dos agentes é o ponto chave da IBSU. Por esse motivo, as seções seguintes se dedicam à questão da segurança.

Mecanismos de Segurança

A segurança está relacionada à necessidade de proteção contra o acesso ou manipulação, intencional ou não, de informações confidenciais por elementos não autorizados. Para o desenvolvimento de um sistema

seguro deve-se levar em consideração as seguintes propriedades (Chin, 1999):

Confidencialidade: apenas as entidades envolvidas podem ter acesso ao conteúdo dos dados que estão trafegando na rede.

Integridade: deve-se garantir que a informação transmitida em um ponto é a mesma recebida em outro.

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 a criptografia, a assinatura digital e a autorização.

Criptografia

A criptografia (Coulouris et al. 1994) 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.

Para tornar o mecanismo de criptografia mais seguro utiliza-se uma chave, 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.

A criptografia é utilizada para garantir a propriedade de confidencialidade, mas não garante a autenticidade do emissor da mensagem. Para isso, deve-se utilizar a criptografia juntamente com as assinaturas digitais.

A criptografia pode utilizar-se de chave secreta ou pública para codificar uma mensagem.

Assinatura Digital

A criptografia é utilizada para implementar o mecanismo de assinatura digital (Coulouris et al. 1994) (Bhimani, 1996). 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.

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.

A figura 2 ilustra o processo de geração e verificação de uma assinatura digital juntamente com criptografia.

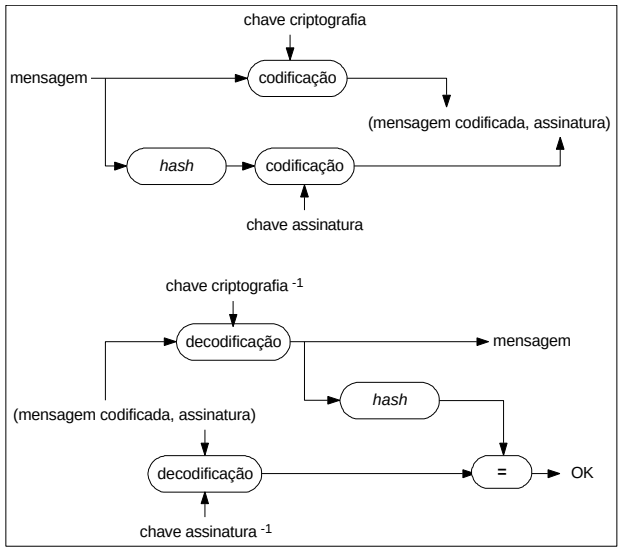

Figura 2 - Geração e Verificação de Assinatura Digital

De acordo com a figura 2, 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.

A assinatura é enviada pela rede juntamente com a mensagem codificada. No destino, acontece o processo inverso. 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.

Considerando a figura 2, 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. Tal garantia é obtida através de certificados e autoridades certificadoras (Chin, 1999).

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.

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.

Quando implementadas juntamente com certificados, as assinaturas digitais servem para garantir as propriedades de integridade, autenticação, controle de acesso e não repudiação (Chin, 1999). Para garantir também a confidencialidade, a mensagem a ser enviada deve estar criptografada, como ilustrado na figura 2.

Autorização

Os mecanismos de autorização (Coulouris, 1994) estão diretamente relacionados ao controle de acesso, 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 ao 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 Segurança de Java

Java possibilita o desenvolvimento de programas seguros. Isso significa que Java previne falhas acidentais nos programas, pois possui endereçamento automático de ponteiros, liberação automática de memória (Garbage Collection), controle de casting, verificação de byte codes e class loading (Zukowski e Rohaly, 2000). Mas, o aspecto de segurança de Java mais importante é de responsabilidade da classe SecurityManager.

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, podendo com isso restringir ou liberar recursos para uma determinada aplicação Java (Meloan, 1999) (Gong, 2000). Uma aplicação pode possuir um domínio de proteção, que é formado por um conjunto de permissões (Sun, 1998b). Como ilustra a figura 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. 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.

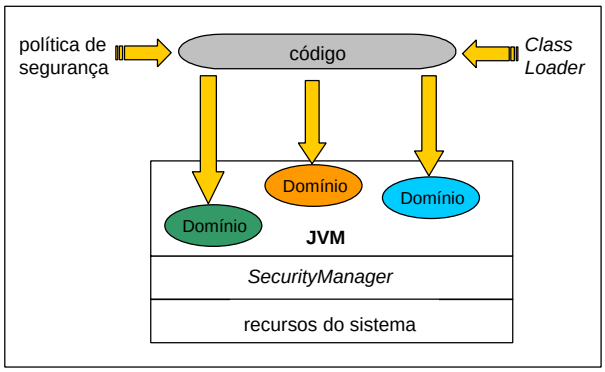

Figura 3 - Modelo de segurança do JDK 1.2

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 (Sun, 1998c). São os policy files que especificam as permissões associadas a cada aplicação ou entidade.

Um policy file pode ser criado através de um editor simples ou através da ferramenta gráfica policytool (Dageforde, 2000) (Sun, 2000), incluída no JDK 1.2. O policy file pode conter várias entradas, cada uma delas 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.

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, como descrito mais adiante.

Java 2 conta com uma API de segurança (Sun, 1999) e com as ferramentas keytool e jarsigner para promover assinaturas digitais (Dageforde, 2000) (Sun, 2000). A API possibilita a assinatura de dados e sua verificação, enquanto as ferramentas possibilitam a assinatura e verificação de aplicações envolvendo certificados. Utilizando-se a API em conjunto com as ferramentas, pode-se garantir a autenticidade do dono de uma aplicação.

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.

Um Java ARchieve ou simplesmente JAR (Sun, 2000) (Sommerer, 2000) 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.

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.

A Ferramenta Keytool

Keytool é uma ferramenta para gerenciar chaves e certificados e armazená-los em keystores, que 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.

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 Jarsigner

A ferramenta jarsigner gera assinaturas digitais para um arquivo JAR e as verifica.

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 armazenados em um keystore para gerar a assinatura.

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. 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 4 ilustra o processo para gerar a assinatura digital de um JAR.

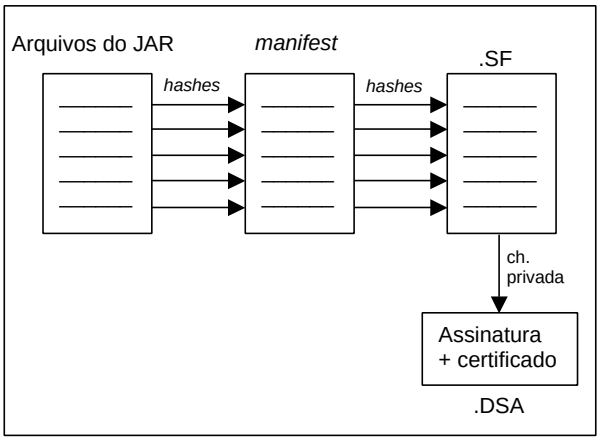

Figura 4 - Assinatura de um JAR

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 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-los 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.

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.

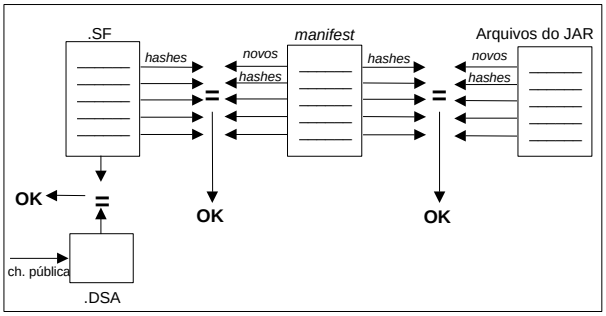

Figura 5 - Verificação da assinatura de um JAR

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.

As Ferramentas de Java 2 Utilizadas em Conjunto

A linguagem Java fornece todos os recursos para garantir a segurança de aplicações. A figura 6 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 6, 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. As ferramentas keytool e jarsigner, quando utilizadas em conjunto, fornecem uma maneira simples e eficiente de tornar as aplicações Java confiáveis.

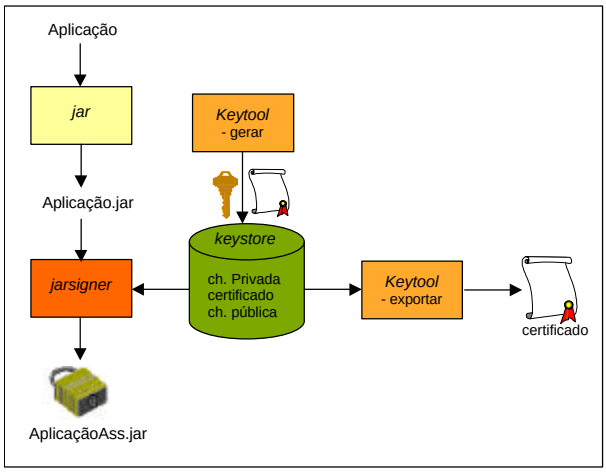

Figura 6 – Processo para Assinatura de Aplicações

A figura 7 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.

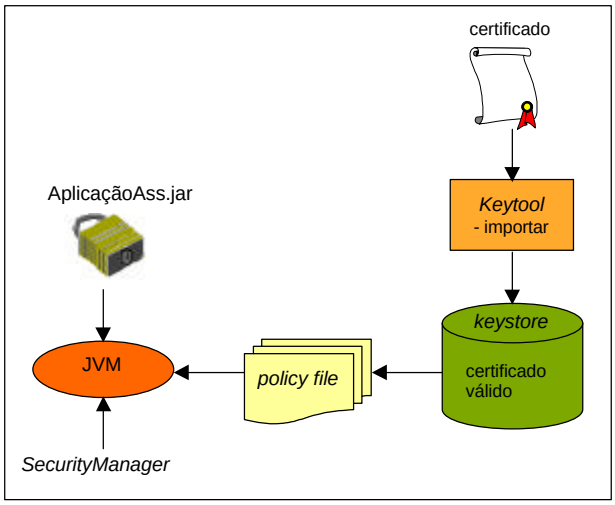

Figura 7 – Autenticação e Execução de Aplicações

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.

Implementação da Interface

A figura 1 apresenta uma visão geral da Interface Básica para um Servidor Universal (IBSU) desenvolvida. As seções seguintes tem por objetivo descrever com maiores detalhes cada um dos módulos da IBSU.

O Agente Gateway

O Gateway é um agente que faz parte da IBSU. Os agentes só poderão entrar no Pool 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. Para cumprir a sua tarefa, o agente Gateway possui quatro classes principais, como ilustra a figura 8.

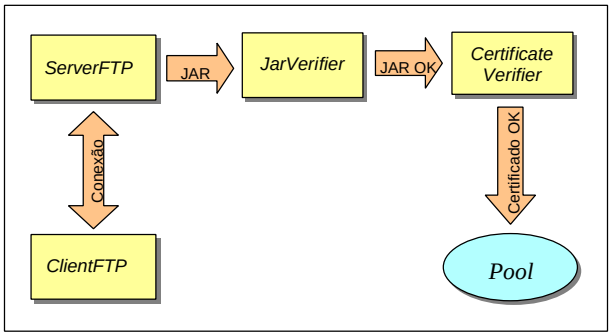

Figura 8 - O Gateway

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. 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. 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.

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. 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.

É importante observar que, para um agente ser reconhecido e verificado com sucesso pelo Gateway, ele deve satisfazer aos seguintes requisitos:

1. Ser desenvolvido em Java.

2. Suas classes devem ser embutidas em um arquivo JAR. 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.

3. Estar devidamente assinado. 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.

4. A entidade que o assinou deve fazer parte de um grupo válido armazenado no banco de dados.

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, 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. 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. 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.

A IBSU utilizou um policy file para promover a segurança. Esse policy file possui duas entradas, uma associada ao grupo superusers e a outra associada ao grupo commonusers. Para os superusers foram dadas todas as permissões possíveis. 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 SecurityException será gerada.

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 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.

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 9.

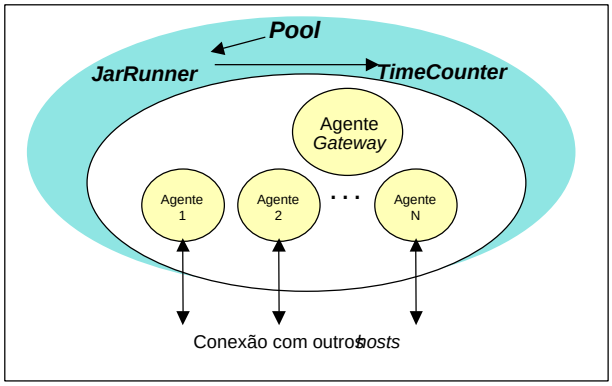

Figura 9 - O Pool de Agentes

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. A classe JarRunner irá começar a execução de um agente logo depois que o Gateway terminar sua verificação.

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 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.

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.

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 na tabela 1.

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. 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.

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

Conclusões

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.

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

Tabela 1 - Descrição dos métodos definidos na Interface com o Banco de Dados

	Método	Funcionalidade
Manipulação de grupos	public void initialize()	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.
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.
	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.
	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
Controle de threads	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. É chamado na classe JarRunner 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.
	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.

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.
- 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.

- Controle de acesso aos roots do banco de dados, o que é garantido na implementação dos métodos definidos na Interface com o Banco de Dados.

Conclui-se 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.

Referências Bibliográficas

A. Bhimani, "Securing the commercial internet", *Communications of the ACM*, 39:6 (1996), 29-35.

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

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

D. Wong, N. Paciorek, D. Moore, "Java-based mobile agents", *Communications of the ACM*, 42:3 (1999), 92-102.

G. Coulouris, J. Dollimore, T. Kindberg, *Distributed systems: concepts and design*, Addison-Wesley, 1994.

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

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

L. Gong, *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>

M. Dageforde, *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>

P. Maes, "Artificial life meets entertainment: lifelike autonomous agents", *Communications of the ACM*, 38: 11 (1995), 108-114.

S. Franklin, A. Graesser, "Is it na agent or just a program?: a taxonomy for automonous agents", *In: Third International workshop on agent theories, architecture and languages*, 1996. [online]. [14/01/2000]. Disponível na Internet: <http://www.msci.memphis.edu/~franklin/AgentProg.html>

S. K. Chin, "High-confidence design for security", *Communications of the ACM*, 42:37 (1999), 33-37.

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

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>

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>

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>

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>

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>

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>