UNIVERSIDADE FEDERAL DO MARANHÃO
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA DE ELETRICIDADE
CURSO DE PÓS-GRADUAÇÃO EM ENGENHARIA DE ELETRICIDADE

FALKNER DE ARÊA LEÃO MORAES

SEGURANÇA E CONFIABILIDADE EM IDS BASEADOS EM AGENTES

São Luís
2009

FALKNER DE ARÊA LEÃO MORAES

SEGURANÇA E CONFIABILIDADE EM IDS BASEADOS EM AGENTES

Dissertação apresentada ao Curso de Pós-Graduação em Engenharia de Eletricidade da Universidade Federal do Maranhão para obtenção do título de Mestre em Engenharia de Eletricidade, área de Concentração: Ciência da Computação.

Orientador: Ph.D. Zair Abdelouahab

São Luís
2009

Moraes, Falkner de Arêa Leão
SEGURANÇA E CONFIABILIDADE EM IDS BASEADOS EM AGENTES / Falkner de Arêa Leão Moraes. – São Luís, 2009.

115f.

Orientador: Zair Abdelouahab.
Impresso por computador (fotocópia).
Dissertação (Mestrado) - Universidade Federal do Maranhão, Programa de Pós-Graduação em Engenharia de Eletricidade. São Luís, 2009.

1. Sistemas Multiagentes. 2. Segurança. 3. IDS. I. Abdelouahab, Zair, orient. II. Título.

CDU 004.891

SEGURANÇA E CONFIABILIDADE EM IDS BASEADOS EM AGENTES

Falkner de Área Leão Moraes

Dissertação aprovada em 16 de fevereiro de 2009.

Prof. Zair Abdelouahab, Ph.D.
(Orientador)

Prof. Djamel Pawzi Hadj Sadok, Ph.D.
(Membro da Banca Examinadora)

Prof. Denivaldo Cicero Pavão Lopes, Dr.
(Membro da Banca Examinadora)

A Deus e ao meu pai Antonio Ferreira de Moraes (*in memoriam*).

Agradecimentos

A DEUS, pela minha vida e pela sua infinita presença em todas as minhas realizações.

À minha mãe Maria dos Humildes Moraes, que sempre permaneceu do meu lado e meu deu apoio incondicional diante das minhas decisões.

Às minhas irmãs, Rakynelly Moraes e Rakylenny Moraes, e ao meu amigo e cunhado Emmanuel Ribeiro.

Ao meu orientador, Prof. Ph.D. Zair Abdelouahab, pelo apoio e confiança depositado em mim.

Ao Prof. Dr. Denivaldo Lopes, pelos incentivos e direcionamentos fundamentais nos momentos de dúvida.

A Profª. M.Sc. Maria Auxiliadora Freire, que me abriu as portas e me lançou nesse desafio.

Aos Profs. Dr. Aristófanes Corrêa, Dr. Mário Meireles e Dra. Maria da Guia Silva.

À minha tia Maria do Carmo Maia, por acreditar e mim e pelo apoio moral.

Ao meu amigo Emerson Oliveira, pela assistência fundamental dada no começo deste trabalho.

Aos meus grandes amigos de laboratório, Helaine Sousa, Aline Lopes, Flávio Ramos e Johnneth Fonsêca, que me acompanharam de perto no desenvolvimento deste trabalho.

Aos meus amigos; Alcides Neto, Irlandino Almeida, Ivo Serra, Geysa Chaves, Gilberto Cunha, Osvaldo Junior, Ricardo Ataíde, Adriana Leite e Paulo Correa.

E a todos que, direta ou indiretamente, contribuíram para a realização desse trabalho.

“Conhecer não é demonstrar nem explicar, é aceder à visão.”

Antoine de Saint-Exupéry

Resumo

A falta de segurança é uma preocupação constante em sistemas distribuídos abertos. Ameaças estão presentes dentro de ambientes inseguros, incertos e que mudam constantemente. Devido a esses problemas, diversas ferramentas para avaliação de vulnerabilidades da rede, bem como para sua proteção, estão sendo desenvolvidas como técnicas de criptografia e softwares como antivírus, *firewall* e IDS (*Intrusion Detection System*). Dentre estas, destaca-se Sistemas IDS que estão crescentemente sendo concebidos, projetados e implementados, usando técnicas de segurança executadas por agentes. Entretanto, é necessário que a segurança e a confiabilidade das mensagens trocadas dentro de um sistema IDS sejam asseguradas. Para este fim, este trabalho propõe uma solução segura e confiável para IDS baseada em agentes. A solução propõe estabelecer um esquema de execução e comunicação segura dos agentes através de mecanismos de proteção de informações de configuração, autenticação e autorização, controle de chaves e persistência de mensagens do IDS, utilizando XML. A solução proposta é implementada como uma extensão do IDS-NIDIA *(Network Intrusion Detection System based on Intelligent Agents)*, cuja arquitetura consiste em uma sociedade de agentes inteligentes que se comunicam de forma cooperativa em um ambiente distribuído. A implementação do protótipo e os testes apresentados neste trabalho demonstram a aplicabilidade da solução proposta.

Palavras-chave: Segurança, IDS, Sistemas Multiagentes, Criptografia, Autenticação, Autorização, Timestamp, Persistência.

Abstract

Lack of security is a constant concern in open distributed systems. Threats are present within environments insecure, uncertain and constantly changing. Due to this problem, many tools for evaluating vulnerabilities of the network as well as for their protection are being developed as techniques for encryption and software systems such as antivirus, firewall and IDS (*Intrusion Detection System*). Among these, there are IDS systems that are being conceived, designed and implemented, using techniques executed by agents. However, it is necessary to assure security and reliability of exchanged messages inside IDS. For this purpose, this paper proposes a security solution for IDS based on agents. The proposed solution provides a methodology and a secure mechanism for communication among agents, through information protection configuration mechanisms, authentication and authorization, key control and messages persistence using XML. The proposed solution is implemented as an extension to the IDS-NIDIA (Network Intrusion Detection System based on Intelligent Agents), whose architecture has an intelligent agent society that communicate in a cooperative way in a distributed environment. The implementation of the prototype and tests proposed in this work show the applicability of the proposed solution.

Keywords: Security, IDS, Multiagent System, Cryptography, Authentication, Authorization, Timestamp, Persistence.

Lista de Figuras

Pg.

Lista de Listagens

Lista de Siglas

AA	Authentication Authority
ABAC	Attribute based Access Control
ACC	Agent Communication Channel
ACL	Agent Communication Language
AMS	Agent Management System
API	Application Programming Interface
ARP	Address Resolution Protocol
CA	Certification Authority
CIDF	Common Intrusion Detection Framework
CIDS	Cougaar based Intrusion Detection System
DAC	Discretionary Access Control
DES	Data Encryption Standard
DF	Directory Facilitator
DFDB	Data Forensic Data Base
EMS	Enterprise Message Systems
FIPA	Foundation for Intelligent Physical Agents
FTP	File Transfer Protocol
GUI	Graphical User Interface
GUID	Globally Unique Identifier
HIDS	Host Intrusion Detection System
HSA	Host Sensor Agent
HTTP	HyperText Transfer Protocol
IBAC	Identification based Access Control
IDE	Integrated Development Environment
IDS	Intrusion Detection System
IIDB	Intruder Intrusion Data Base
IP	Internet Protocol
J2EE	Java 2 Platform, Enterprise Edition
JADE	Java Agent DEvelopment Framework
JMS	Java Message Service
JNDI	Java Naming and Directory Interface
JVM	Java Virtual Machine
LBAC	Label based Access Control
LSIA	Local Security Intelligent Agent
MAC	Message Authentication Code
MAC	Mandatory Access Control
MAC	Media Access Control
MCA	Main Controller Agent
MIDS	Multi-level and multi-agent Intrusion Detection System
MOM	Message Oriented Middleware
MQI	Message Queue Interface
MQM	Message Queue Manager

MTP	Message Transport Protocol
NIDIA	Network Intrusion Detection System based on Intelligent Agents
NIDS	Network Intrusion Detection System
NSA	Net Sensor Agent
OOM	Object Oriented Model
OSI	Open Systems Interconnection
PIN	Personal Identifier Number
PKI	Public Key Infrastructure
RADB	Response Action Data base
RBAC	Rule based Access Control
RDF	Resource Description Framework
RMI	Remote Method Invocation
RPC	Remote Procedure Call
RSA	Rivest-Shamir-Adleman
SCA	System Control Agent
SFTP	SSH File Transfer Protocol
SEA	System Evaluation Agent
SMA	System Monitoring Agent
SOAP	Simple Object Access Protocol
SSH	Secure Shell
SSL	Secure Socket Layer
SSO	Single Sign-On
STDB	STrategy Data base
SUA	System Update Agent
TCP	Transport Control Protocol
TLS	Transport Layer Security
TSIK	Trust Service Integration Kit
UML	Unified Modeling Language
TTP	Truted Third Part
W3C	World Wide Web Consortium
XKMS	XML Key Management Specification
XML	eXtensible Markup Language
ZK	Zero-Knowledge

Sumário

Pg.

1 Introdução

Ao longo dos anos, os sistemas de computadores têm evoluído bastante e com sucesso, passando de meros dispositivos de computação monolítica (suportando aplicações estáticas), para grandes redes de computação distribuída. Entretanto, essa evolução trouxe consigo uma série de novos desafios, tornando a segurança de sistemas distribuídos fundamentalmente mais complexa. Em outras palavras, estes sistemas, pela sua arquitetura, tornaram-se cada vez mais expostos a interações maléficas, sujeitando-se a um conjunto de ameaças a sua segurança, obrigando o desenvolvimento de novos mecanismos que sejam adequados para oferecer serviços de proteção.

Segurança é uma preocupação constante em sistemas distribuídos abertos. Por conta disso, a necessidade de introduzir mecanismos de proteção é um fato que vem transcendendo o limite da produtividade e da funcionalidade, e surge naturalmente quando se processam informações sensíveis dentro de ambientes inseguros, incertos e que mudam constantemente. Enquanto a velocidade e a eficiência em todos os processos de negócios significam uma vantagem competitiva, a falta de segurança nos meios que nela habitam pode resultar em grandes prejuízos. Sendo assim, uma rede de computadores necessita contar com mecanismos de segurança e políticas de segurança desenvolvidos sob sistemas operacionais subjacentes e implementados como ferramentas de segurança como antivírus, firewall e IDS (*Intrusion Detection System*).

Ferramentas de segurança são sistemas de software complexos, pois envolvem estatísticas, heurísticas, reconhecimento de padrões e, mais recentemente, agentes de software. Agentes de software possuem um modelo que é adequado para a construção de componentes autônomos que agem e interagem de modo flexível para alcançar os objetivos de segurança em ambientes inseguros, incertos e dinâmicos [RAMCHURN, 2005].

Dentro desse contexto, um elevado grau de importância é dado às ferramentas de segurança IDS. Estas ferramentas vêm ganhando destaque pela sua crescente concepção, projeção e implementação através do uso de técnicas de segurança executadas por agentes. Com o rápido desenvolvimento das tecnologias de rede e de computadores, novas formas de intrusão acabam emergindo na mesma velocidade. Um ambiente marcado pela evolução contínua faz com que novos ataques tenham como resposta novas formas de proteção, que levam ao desenvolvimento de novas técnicas de ataque, de maneira que um ciclo de ataque e defesa é formado. Portanto, os sistemas de detecção devem ser capazes de serem atualizados e

constantemente serem melhorados, utilizando meios inteligentes e dinâmicos para se adaptar ao novo ambiente.

Como exemplo desses avanços, tem-se o projeto IDS-NIDIA (*Network Intrusion Detection System based on Intelligent Agents*) [LIMA, 2001] [SILVA, 2006] [OLIVEIRA, 2006], que propõe um sistema de detecção de intrusão em tempo real, capaz de monitorar através de uma sociedade de agentes inteligentes, se uma rede está sofrendo acessos não autorizados que podem indicar a ação de entidades externas (como *hackers*) ou até mesmos indivíduos internos mal intencionados. Esse modelo de IDS foi idealizado com o propósito de fornecer uma contribuição para a melhoria das técnicas de detecção de intrusos por meio de componentes inteligentes, autônomos e cooperativos.

1.1 Descrição do Problema

No mundo da informação digital, invasões e outros tipos de ataques direcionados às grandes redes de computação distribuída tornam-se cada vez mais generalizados e sofisticados, forçando a adoção de uma segurança contínua e evolutiva. Um arsenal de defesa que antes funcionava contra determinados tipos de ataques, pode ser falho contra novas técnicas desenvolvidas para driblar esse mesmo arsenal de defesa.

Uma série de mecanismos e tecnologias de segurança tem sido aplicada em defesa dos recursos existentes numa rede distribuída. Na vanguarda contra essas ameaças têm-se os antivírus, firewalls, criptografia, procedimentos de autenticação, mecanismos de controle de vulnerabilidades e outras medidas que visam oferecer segurança aprimorada. Entretanto, com todos esses elementos, grande parte dos sistemas possui algum tipo de vulnerabilidade originado por problemas de implementação, configuração ou projeto, que poderão ser futuramente explorados para os mais diversos fins, como meio de negação de serviço ou até mesmo permitir um acesso não autorizado ao sistema. Para conter esse problema, uma segunda linha de defesa conhecida como Sistema de Detecção de Intrusão (IDS) foi criada para detectar atividades hostis em uma rede e impedir atividades que possam comprometer a segurança de um sistema.

Os sistemas IDS estão se mostrando grandes aliados dos administradores de segurança e, por isso, são consideradas ferramentas essenciais no combate ao uso mal intencionado da rede. Suas técnicas de detecção estão constantemente sendo aprimoradas e recentes pesquisas voltadas à utilização de agentes de software no desenvolvimento desses sistemas vêm proporcionando ganhos significativos em eficiência e escalabilidade, principalmente em

operações que envolvem análise de comportamento (tanto do sistema quanto dos usuários) e realização de contramedidas. No entanto, sistemas de detecção que utilizam agentes para garantir a segurança de redes distribuídas abertas, não necessariamente fazem deles sistemas verdadeiramente seguros de serem utilizados. Tratando-se de sistemas IDS baseado em agentes, a necessidade de garantir a segurança dos próprios componentes desse sistema, durante sua execução e comunicação, transcende a necessidade de proteger a rede do qual eles fazem parte.

Os seguintes fatores podem ser considerados para que a preocupação com a segurança dos agentes de um sistema IDS seja justificada [NAKAMURA, 2007]:

- **Entender a natureza dos ataques é fundamental**. Muitos dos ataques são resultados da exploração de vulnerabilidades, as quais passam a existir devido a uma falha no projeto ou na implementação de um sistema. Portanto, se agentes integram um sistema de segurança, estes devem assegurar-se de não serem os alvos a serem explorados;
- **Novas tecnologias trazem consigo novas vulnerabilidades**. Novas tecnologias e novos sistemas estão constantemente sendo criados e junto com elas surgem novas vulnerabilidades. Por isso, é natural considerar que o uso da tecnologia de agentes não é a exceção. As vulnerabilidades também estarão presentes e, por conseguinte, novos ataques também serão criados;
- **A existência de ataques direcionados a sistemas de segurança**. Ataques direcionados a sistemas de segurança podem ser considerados mais perigosos, pois, existindo a intenção de atacar, a estratégia pode ser cuidadosamente pensada e estudada, e executada de modo a explorar o elo mais fraco desse sistema. No caso de IDS baseado em agentes, a execução e a comunicação desses agentes podem perfeitamente representar esse elo mais fraco;
- **A defesa é mais complexa que o ataque**. Para um intruso, apenas um ponto de falha do sistema é suficiente para ser explorado. Caso uma determinada técnica utilizada na invasão não funcione, ele pode explorar outras, até que os seus objetivos sejam atingidos. No entanto, para um sistema de segurança, a defesa é muito mais complexa, pois exige que todos os pontos de falha dentro da sua estrutura interna estejam protegidos.

Os fatores descritos acima são suficientes para concluir que a segurança de um sistema IDS deve ser assegurada. A necessidade de desenvolver mecanismos de defesa que corrijam as falhas contidas na estrutura interna desses sistemas, privilegiando a execução e a comunicação segura dos seus componentes, é a única forma de torná-los menos suscetível a ameaças que possam comprometer o papel principal desses sistemas que é garantir a segurança da rede.

Baseado na estrutura interna do IDS-NIDIA, os seguintes problemas foram identificados:

- Falta de um mecanismo dedicado a proteção de informações sensíveis armazenadas em disco, que são necessárias para a execução dos agentes;
- Carência de mecanismos de autenticação e controle de acesso ao servidor de chaves para que os agentes possam armazenar e utilizar as chaves públicas do sistema;
- Falta de um sistema de controle de validade das chaves públicas utilizadas na codificação das mensagens de comunicação entre os agentes;
- Ausência de garantia de entrega das mensagens entre os agentes, caso haja uma possível falha do receptor.

As mensagens de comunicação entre os agentes são transmitidas utilizando um protocolo que define um conjunto de informações que passam por um processo de padronização para serem repassadas. Sendo assim, o termo mensagem definido no decorrer desta dissertação refere-se a toda e qualquer informação enviada ou recebida através de um canal de comunicação que segue regras e formatos de padronização bem definidos, constituindo um protocolo de comunicação seguro e confiável.

1.2 Metodologia

A metodologia empregada para o desenvolvimento deste trabalho teve como ponto de partida a geração de conhecimentos, que utilizou inicialmente a pesquisa bibliográfica para atingir os objetivos propostos. Em especial, a pesquisa reuniu um conjunto de informações de livros, teses, dissertações, artigos científicos e documentos hipermídia úteis para a investigação do problema. Esta fase preliminar tende a obter uma completa familiarização com o contexto da literatura especializada.

Em seguida, fez-se um levantamento das informações consideradas essenciais para a elaboração da solução e uma análise de ferramentas que pudessem auxiliar no processo de modelagem e implementação dos mecanismos de segurança. A solução proposta teve seu protótipo modelado e projetado utilizando a Metodologia de Análise e Projeto Orientado a Objetos e foi implementado como uma API utilizando a linguagem Java.

Com base nestes estudos, adaptação de ferramentas e integração das bibliotecas selecionadas, a solução de segurança e confiabilidade foi construída, adaptada e estendida à estrutura interna de sistemas IDS baseado em agentes. Especificamente, mecanismos de segurança foram desenvolvidos baseando-se no esquema utilizado na comunicação e execução dos agentes do IDS-NIDIA.

As seguintes ferramentas foram estudas e utilizadas no desenvolvimento da solução:

- Java Eclipse IDE (*Integrated Development Environment*) que se constitui em um ambiente para a integração de ferramentas de desenvolvimento;
- JADE (*Java Agent Development Framework*) [JADE, 2008] que simplifica a implementação de sistemas multiagentes através de um "*middleware*" que cumpre as especificações FIPA, e utiliza um conjunto de ferramentas gráficas que dão suporte a *debuging* e às fases de desenvolvimento;
- API *Systinet Server* for Java da companhia *Systinet* [SYSTINET, 2006] que é um ambientes independente de plataforma, de fácil utilização e de alto desempenho para a criação e desenvolvimento de *web services* em Java para aplicações J2EE;
- TSIK (*Trust Service Integration Kit*) [VERISIGN, 2008] que é um kit de desenvolvimento baseado em Java e disponibilizado pela *Verisign*, para integrar compatibilidades de segurança em *web services* incluindo características de segurança como assinatura XML, encriptação XML e XKMS (*XML Key Management Specification*);
- OpenJMS [OPENJMS, 2008] que uma implementação de código aberto de uma especificação JMS (*Java Message Service*) que fornece mecanismos de persistência de mensagens;
- Bibliotecas para desenvolvimento de mecanismos de *timestamps*;
- Bibliotecas para desenvolvimento de mecanismos de autenticação e controle de acesso.

1.3 Objetivos Gerais e Específicos

O objetivo geral deste trabalho é propor uma solução segura e confiável para agentes de sistemas IDS. Em outras palavras, pretende-se introduzir melhorias significativas à estrutura interna desses sistemas de detecção, oferecendo mecanismos de proteção à comunicação e à troca de mensagens entre os agentes durante sua execução.

De modo mais específico, o trabalho propõe uma solução de segurança e confiabilidade para IDS, tomando como base às deficiências de proteção de informações de configuração, autenticação e autorização, controle de chaves e persistência de mensagens. Portanto, a solução consiste em estabelecer um esquema de execução e comunicação segura e confiável dos agentes através de:

- Mecanismos de proteção de informações de configuração do sistema (através de algoritmos de criptografia);
- Mecanismos de autenticação e autorização entre os agentes do sistema e o servidor, para armazenamento e recuperação de chaves;
- Mecanismos de controle de validade das chaves públicas contidas no servidor de chaves do sistema (utilizando algoritmos de *timestamps*);
- Mecanismos de persistência, que garantam a entrega das mensagens trocadas pelos agentes (auxiliados por um provedor JMS).

A solução proposta é implementada como uma extensão do IDS-NIDIA, que define uma arquitetura composta de uma sociedade de agentes inteligentes que se comunicam de forma cooperativa em um ambiente distribuído. A modelagem, prototipagem e os testes descritos neste trabalho demonstram a aplicabilidade da solução.

1.4 Organização da Dissertação

O texto da dissertação é uma reflexão sobre as diversas etapas que foram cumpridas para o desenvolvimento da solução proposta e está organizada em seis capítulos. No primeiro Capítulo, um contexto geral em que o trabalho está inserido, bem como, a metodologia empregada e os objetivos da proposta são apresentados.

O Capítulo 2 apresenta a fundamentação teórica da dissertação, expondo conceitos essenciais para compreensão de nossa proposta. Os seguintes conceitos são listados: agentes, sistemas multiagentes, sistemas IDS e a importância dada às recentes implementações destes

utilizando agentes de software, ferramentas MOM (*Message Oriented Middleware*), autenticação e autorização de usuários, e o IDS-NIDIA, juntamente com sua arquitetura e implementação.

O Capítulo 3 apresenta a descrição da solução proposta como modelo de segurança para IDS baseados em agentes. O modelo proposto é constituído de quatro partes: proteção de informações de configuração, autenticação e autorização, gerenciamento de chaves e *timestamps*, e persistência de mensagens.

O Capítulo 4 oferece detalhes da modelagem e prototipagem da solução proposta. Neste capítulo, a solução em termos de seus componentes é apresentada. Diagramas de pacotes, diagramas de classes e diagramas de seqüência do processo de modelagem são mostrados especificamente para cada mecanismo utilizado na solução e a descrição individual desses mecanismos, em termos de código do protótipo desenvolvido para testes dos objetivos alcançados também são acrescentados.

O Capítulo 5 contém a descrição da aplicabilidade da solução proposta integrada ao IDS-NIDIA. Neste capítulo, a implementação do protótipo, estendendo e adaptando o IDS-NIDIA é apresentada. Sendo assim, o desenvolvimento dos componentes pertencentes à solução proposta e os testes que demonstram o funcionamento do sistema, são mostrados.

Finalmente, no Capítulo 6 são discutidos os resultados obtidos e a sua relevância no contexto da segurança de agentes de sistemas que operam em ambientes inseguros. Algumas propostas para trabalhos futuros são apresentadas.

2 Fundamentação Teórica

Este capítulo tem por objetivo apresentar uma visão geral dos conceitos fundamentais que formaram as bases para o desenvolvimento deste trabalho. Tais conceitos tiveram papel relevante na compreensão do problema e obtenção do conhecimento necessário ao desenvolvimento da solução.

2.1 Agentes e Sistemas Multiagentes

O significado de agentes utilizando um conceito único ou padrão é muito difícil, pois não existe consenso diante das várias abordagens e ponto de vista de diferentes autores. Além disso, devido às suas mais diversas aplicações, uma definição precisa torna-se cada vez mais complexa e variada.

Embora existam semelhanças entre objetos e agentes, as diferenças entre eles são bastante evidentes. Entre elas pode-se citar o fato de agentes terem uma noção mais forte de autonomia em relação aos objetos, além de serem capazes de um comportamento flexível e inerentemente *multi-thread*.

No geral, os agentes são entendidos como entidades contidas em um ambiente e são capazes de sentir e agir (não necessariamente nessa ordem) [BELLIFEMINE, 2007]. Sendo assim, eles não são considerados entidades isoladas, pois são capazes de se comunicar e colaborar com outras entidades. Entre os atributos que podem estar presentes e que melhor caracterizam um agente incluem [OLIVEIRA, 2006]:

- **Autonomia**. O agente toma decisões e ações importantes para a conclusão de uma tarefa ou objetivo sem a necessidade da interferência humana ou de qualquer outra entidade;
- **Interatividade**. O agente comunicar-se com o ambiente e outros agentes;
- **Adaptação**. O agente modifica o seu comportamento baseado em experiências com o seu ambiente ou com outros agentes;
- **Sociabilidade**. O agente não é uma entidade isolada e, portanto, realiza interações sociáveis com outros agentes;
- **Mobilidade**. O agente move-se seja por uma rede local (*intranet*) ou até mesmo, pela Internet, transportando-se pelas plataformas levando dados e códigos;
- **Proatividade**. O agente é capaz de tomar iniciativas, exibindo seu comportamento baseados em objetivos;

- **Inteligência**. O agente adquire conhecimentos e as transmite aos outros agentes usando linguagem de símbolos;
- **Racionalidade**. O agente tem a capacidade de analisar e inferir baseando-se no conhecimento atual e nas suas experiências;
- **Cooperatividade**. Os agentes trabalham juntos para mútuo benefício na execução de uma tarefa complexa e um comportamento adaptativo, os quais podem examinar o meio externo e adaptar suas ações para aumentar a probabilidade de ser bem sucedido em metas.

Apesar de não existir uma definição consensual do conceito de agente, existe a noção de que um agente de software é uma entidade autônoma que pode interagir com o ambiente. Agentes de software são implementados através de software e podem reagir com outras entidades como humanos, máquinas e outros agentes em vários ambientes, através de várias plataformas. Eles são aplicáveis nos mais diversos meios como [SILVA, 1999]:

- **Processo de Ensino-aprendizagem**. Utilização de agentes no aprendizado e no processo de treinamento de usuários;
- **Indústria**. Os sistemas que atuam neste propósito são complexos e isolados. Portanto, a vantagem da aplicação de agentes na produção industrial seria integrar e compartilhar informações entre esses sistemas;
- **Simulação**. Os agentes simulam situações dando um alto grau de veracidade nas áreas de entretenimentos, pesquisas tecnológicas e militares;
- **Realidade Virtual**. Os agentes atuam como participantes que auxiliam e monitoram certas atividades e usuários, assistindo-os ou ajudando-os quando necessário, principalmente em ambientes virtuais muito complexos;
- **Prestação de Serviços**. Nesse contexto os agentes agregam valores a certos serviços, gerenciando a informação a fim de satisfazer as necessidades dos clientes;
- **Rede de Computadores**. Nesse caso, a função dos agentes varia desde definição de melhores rotas, controle de manutenção de equipamentos, gerenciamento de dados até o monitoramento de atividades e gerência da própria rede.

Os agentes constituem um paradigma de software apropriado para desenvolver aplicações para ambientes abertos, heterogêneos e distribuídos. A adequação dos agentes a

processos de resolução de problemas cuja perspectiva centralizada não demonstra ter capacidade de resolvê-los satisfatoriamente é uma razão para o crescimento relacionado à sua investigação. No campo das pesquisas, a área de investigação designada por agentes autônomos surgiu inspirada nas seguintes áreas científicas:

- **Inteligência Artificial**. Micro-aspectos como a resolução de problemas, raciocínio lógico, representação e utilização de conhecimentos, planejamento e aprendizagem;
- **Engenharia de Software**. O agente como uma abstração, programação orientada por agentes;
- **Sistemas Distribuídos e Redes de Computadores**. Arquiteturas de agentes, Sistemas Multiagentes, comunicação e coordenação;
- **Sociologia**. Macro-aspectos como a formação de sociedades virtuais e a interação entre agentes;
- **Teoria de Jogos e Economia**. Negociação, resolução de conflitos, mecanismos de mercado.

Um Sistema Multiagente é caracterizado pela existência de agentes que interagem de forma autônoma e trabalham juntos para resolver um determinado problema ou alcançar um objetivo comum [BELLIFEMINE, 2007]. Em suma, pode-se dizer que Sistemas Multiagentes são constituídos de múltiplos agentes que interagem ou trabalham em conjunto de forma a realizar um determinado conjunto de tarefas ou objetivos. Esses objetivos podem ser comuns a todos os agentes ou não. Os agentes dentro de um sistema multiagente podem ser heterogêneos ou homogêneos, colaborativos ou competitivos. A definição dos tipos de agentes depende da finalidade da aplicação que o sistema multiagente está inserido.

Sistemas multiagentes podem constituir-se de um grande número de elementos conhecidos como agentes reativos. Estes, pela sua definição, são bastante simples, não possuem inteligência ou representação de seu ambiente e interagem utilizando um comportamento de ação/reação. A inteligência só é identificada através do comportamento global do grupo de agentes.

Já sistemas multiagentes que se constituem de agentes cognitivos são geralmente compostos por uma quantidade bem menor de agentes se comparado aos sistemas multiagentes reativos. Conforme a definição de agentes cognitivos, estes são inteligentes e contém uma representação parcial de seu ambiente e dos outros agentes. Podem, portanto,

comunicar-se entre si, negociar uma informação ou um serviço e planejar uma ação futura. Esse planejamento de ações é possível, pois em geral os agentes cognitivos são dotados de conhecimentos, competências, intenções e crenças, o que lhes permite coordenar ações visando à resolução de um problema ou a execução de um objetivo.

Atualmente, a pesquisa em sistemas multiagentes está interessada principalmente na coordenação das ações e comportamentos dos agentes, como eles coordenam seu conhecimento, planos, objetivos e crenças com o objetivo de tomar ações ou resolver problemas. A razão desse interesse está na capacidade de fornecer robustez e eficiência, permitir interoperabilidade entre sistemas legados e resolver problemas cujo dado, especialidade ou controle é distribuído.

2.1.1 Framework JADE

O JADE (*Java DEvelopment Framework*) [BELLIFEMINE, 2007] é um ambiente *open source* para desenvolvimento de aplicações baseado em agentes que atende as especificações da FIPA (*Foundation for Intelligent Physical Agents*), para interoperabilidade entre sistemas multiagentes totalmente implementado em Java. Muitos autores o consideram como um *middleware*[1], por oferecer um modelo que pode ser utilizado como base para implementação de outros modelos de mais alto nível. Sua estrutura fornece um *framework* completo para tratamento da comunicação, do ciclo de vida e do monitoramento da execução dos agentes, entre outras atividades.

O principal objetivo do JADE é facilitar o desenvolvimento de sistemas multiagentes, garantindo um padrão de interoperabilidade, e tornar possível a comunicação entre eles, através de um conjunto abrangente de agentes de serviços de sistema, conforme as especificações da FIPA [FIPA, 2008]. Sendo assim, o JADE abstrai ao programador muito das especificações da FIPA, tais como:

- Não há necessidade de implementar a plataforma de agentes;
- Não há necessidade de implementar um gerenciador de agentes;
- Não há necessidade de implementar transporte de mensagens e *parsing* (análise gramatical das mensagens);
- Protocolos de interação só podem ser herdados por meio de métodos específicos.

---

[1]middleware é um programa de computador que é executado entre outros softwares.

A Figura 2.1 representa o modelo geral da arquitetura de agentes com o *framework* JADE.

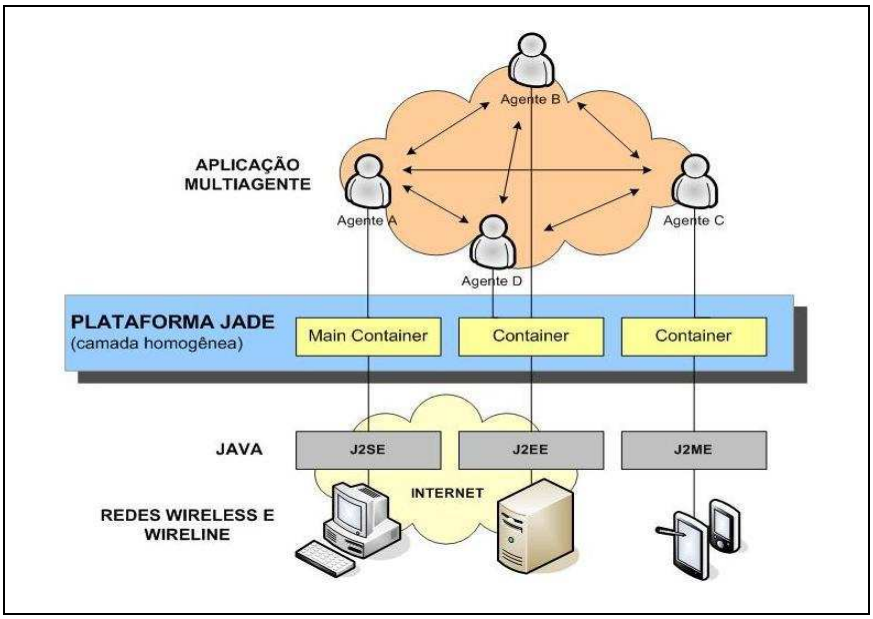

Figura 2.1 – Modelo geral da arquitetura de agentes com o JADE [BELLIFEMINE, 2007].

Entre as características que o JADE oferece para a programação de sistemas multiagentes destacam-se [BELLIFEMINE, 2007]:

- **Plataforma Distribuída de Agentes**. O JADE pode ser dividido em vários *hosts* ou máquinas. Apenas uma aplicação Java e uma JVM (*Java Virtual Machine*) são executadas em cada *host*. Os agentes são implementados como *threads* Java e inseridos dentro de repositórios de agentes chamados de containeres que provêm suporte para execução do agente;
- **GUI (*Graphical User Interface*)**. Interface visual que gerencia vários agentes e containeres de agentes inclusive remotamente;
- **Ferramentas de *Debugging*.** Ferramentas que ajudam o desenvolvimento e depuração de aplicações multiagentes baseados em JADE;
- **Suporte a execução de atividades múltiplas, paralelas e concorrentes de agentes**. Este suporte inclui o sistema gerenciador de agentes (AMS – *Agent Management System*), o diretório facilitador (DF – *Directory Facilitator*) e o canal de comunicação de agentes (ACC – *Agent Communication Channel*);
- **Transporte de Mensagens**. Transporte de mensagens no formato FIPA-ACL dentro da mesma plataforma de agentes;
- **Biblioteca de protocolos FIPA**. Para interação entre os agentes do JADE;

- **Automação de Registros**. Registro e cancelamento automático de agentes com a AMS fazendo com que o desenvolvedor se abstraia dessa tarefa;
- **Serviço de Nomes (*Name Service*) em conformidade aos padrões FIPA**. Obtenção de seus GUID (*Globally Unique Identifier*) da plataforma que são identificadores únicos em todo o ambiente;
- **Integração**. Mecanismo que permite aplicações externas carregarem agentes autônomos do JADE.

A arquitetura da plataforma JADE é baseada na coexistência de várias máquinas virtuais Java (*Java Virtual Machine* – JVM) que são distribuídas por diversas máquinas (*hosts*) independentes do sistema operacional que cada uma utiliza [BELLIFEMINE, 2007]. Cada máquina possui uma JVM para enfatizar o conceito de independência de plataforma e cada JVM tem um *container* que fornece um ambiente completo para execução dos agentes, além de permitir que eles rodem concorrentemente no mesmo processador (*host*). Durante a execução, deve existir uma JVM por processador e podem existir vários agentes por JVM. Na Figura 2.2, tem-se uma representação gráfica da plataforma de agentes do JADE dividida em três *hosts*, com a comunicação entre JVMs sendo executada através de uma RMI (*Remote Method Invocation*).

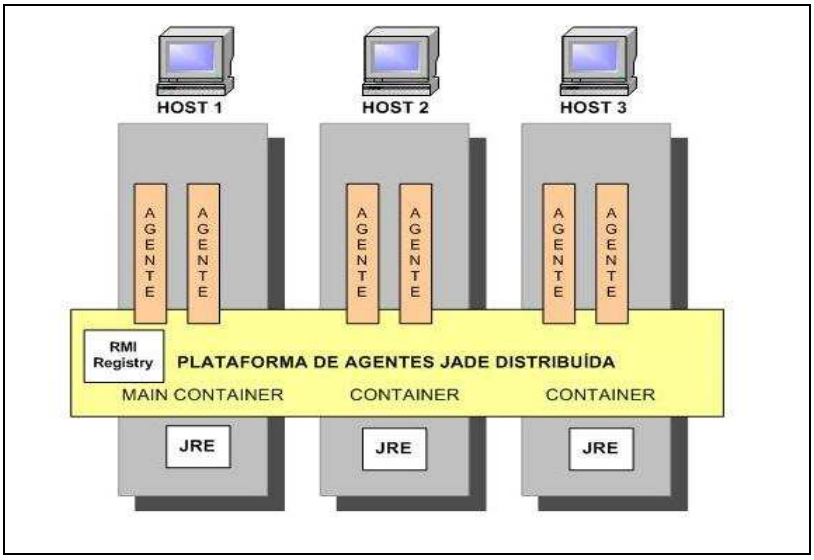

Figura 2.2 – Plataforma de agentes JADE dividida em containeres [BELLIFEMINE, 2007].

Conforme mostrado na Figura 2.2, o *main-container* (localizado no *host* 1) é o container principal onde se encontra o AMS, o DF e o registro RMI (*Remote Method*

Invocation). Esse registro corresponde a um servidor de nomes que o JADE utiliza para manter as referências aos outros containeres de agentes que se conectam a plataforma. Em seguida, têm-se os outros containeres que se conectam ao *main-container*, fazendo com que o desenvolvedor fique abstraído da separação física dos *hosts*, caso exista, ou das diferenças de plataforma dos quais cada *host* possam ter. Essa abstração atua como um elo de ligação e prover um completo ambiente de execução para qualquer conjunto de agentes JADE.

2.2 IDS (*Intrusion Detection System*)

Nos últimos anos, as pesquisas na área de segurança de redes têm se concentrado no desenvolvimento de mecanismos sofisticados de detecção de ataques, que atuam como sentinelas, a espera de alterações no fluxo normal dos processos, acessos não autorizados do sistema e ataques ao fluxo de dados que circulam dentro da rede. Por conta disso, os sistemas IDS estão crescentemente sendo concebidos, projetados e implementados utilizando tecnologias avançadas de combate a esse tipo de ameaça, cujo papel é garantir a integridade da rede.

Antes de definirmos sistemas IDS, é importante saber o que significa uma intrusão. Segundo [DEBAR, 2004], uma intrusão pode ser definida como um conjunto de ações que tenta comprometer a integridade, confidencialidade ou disponibilidade de um recurso. Em outras palavras, considera-se intrusão, toda e qualquer atividade não autorizada ou que não deveria ocorrer em um dado sistema.

Um Sistema de Detecção de Intrusão (*Intrusion Detection System* – IDS) [DEBAR, 2004] é uma ferramenta de segurança que monitora o tráfego de uma rede em busca de sinais de intrusão, atividades de usuários não autorizados e a ocorrência de práticas mal intencionadas, como usuários autorizados que tentam transpassar os limites de restrição de acesso aos recursos. Sendo assim, o objetivo dos sistemas IDS é identificar e impedir a ação tanto de indivíduos que estão utilizando o sistema sem autorização (ameaças externas) como também de indivíduos com acesso legítimo ao sistema, mas que cometem abuso desses privilégios (ameaças internas).

Para uma ferramenta de segurança ser considerada um IDS, é desejável que ela possua as seguintes características [DEBAR, 2004]:

- Deve estar continuamente em execução com um mínimo de supervisão;
- Deve recuperar-se de possíveis falhas ou problemas com a rede;

- Deve analisar a si mesmo e verificar se a sua estrutura interna foi modificada por um atacante;
- Deve utilizar o mínimo de recursos possíveis;
- Deve está configurado de acordo com as políticas de segurança seguidas pela organização;
- Deve se adaptar às mudanças do sistema e usuários, e deve ser facilmente atualizável.

Os sistemas IDS detectam as vulnerabilidades da rede analisando o conteúdo dos pacotes IP que nela trafegam. A detecção desses ataques é baseada em assinaturas contidas no cabeçalho e/ou na carga útil do pacote IP. Essas assinaturas, que constitui uma das limitações do sistema, devem estar sempre atualizadas para se ter sucesso na descoberta do ataque e suas variantes.

2.2.1 Tipos de IDS

O esquema de classificação de um IDS é determinado segundo o tipo de detecção, o modelo de detecção ou o tipo de reação quando este detecta um possível ataque [DEBAR, 2004]. De acordo com o tipo de detecção os sistemas IDS podem ser [DEBAR, 2004]:

- **IDS baseado em Rede (*Network Intrusion Detection System* – NIDS)**. Estes IDS analisam o trafego completo da rede, examinando os pacotes individualmente (incluindo todas as diferentes opções que podem coexistir dentro de um pacote) e detectando pacotes maliciosos que foram projetados para não serem detectados por outras ferramentas como *firewalls*. Estes sistemas também podem localizar o *host* que está sendo invadido e identificar os sinais de alerta que são emitidos quando o invasor tenta explorar alguma falha deste *host*. Como exemplo de um NIDS, tem-se o sistema de detecção e prevenção de intrusão de código aberto denominado SNORT [SNORT, 2009], que utiliza uma linguagem dirigida a regras, que combina os benefícios da assinatura, protocolo e anomalia baseado em métodos de inspeção.
- **IDS baseado em *Host* (*Host Intrusion Detection System* – HIDS)**. Estes IDS analisam o tráfego sobre um servidor ou uma estação qualquer, sem se preocupar com o que esta acontecendo em cada *host* da rede e são capazes de detectar

situações como intensas falhas de acesso ou modificações em arquivo considerados críticos.

Sistemas IDS baseado em rede (NIDS) são ferramentas úteis por que detectam acessos indesejáveis através sensores situados em segmentos estratégicos da rede, que são responsáveis pelo monitoramento em busca de atividades suspeitas. Além disso, sua estrutura contém um console que recebe os alarmes desses sensores e executa as contramedidas, dependendo da configuração de reação aos alarmes recebidos. Entretanto esses sistemas têm alguns pontos negativos [DEBAR, 2004]:

- Não examinam o tráfego total da rede, mas somente os segmentos no qual os sensores estão conectados;
- Capazes de gerar congestionamento na rede devido ao grande volume de dados que percorre na rede;
- Não reconhecem se um ataque foi bem sucedido, apenas apontam que um ataque foi iniciado;
- Difíceis de detectar ataques com dados encriptados.

Sistemas IDS baseado em *host* (HIDS) são ferramentas mais potentes, pois são capazes de manter registro de todos os comandos utilizados e ficheiros abertos além de possuírem a menor taxa de detecção de falsos-positivos[1] em relação aos NIDS. Porém, também apresentam inconvenientes como [DEBAR, 2004]:

- Requer instalação na máquina local que se quer proteger, criando uma carga adicional para o sistema;
- A confiança nas capacidades de auditoria e *logging* recaem somente na máquina o qual está instalada.

De acordo com a classificação por detecção, os sistemas IDS são divididos em dois modelos diferentes [DEBAR, 2004]:

- **Detecção por Mau Uso**. Envolve a verificação sobre os tipos ilegais de tráfego da rede, como por exemplo, combinações dentro de um pacote que não poderiam dar legitimidade. Este tipo de detecção pode incluir a intenção de um usuário em

[1]falso-positivo é um falso alarme de ataque gerado por um IDS.

executar programas sem permissão. Os modelos de detecção por mau uso analisam os pontos frágeis dos sistemas que podem ser explorados e descrevem, através de modelos, a seqüência de eventos que serão interpretados pelo IDS;

- **Detecção por Anomalia**. Este modelo se apóia em estatísticas para compreender se um tráfego é considerado normal ou não dentro da rede. Um claro exemplo de uma atividade anormal seria a detecção de tráfego fora do horário de funcionamento ou acesso repetitivo de uma máquina remota. Este modelo de detecção funciona identificando mudanças no padrão de utilização ou comportamento do sistema. Esta detecção é obtida através de um modelo estatístico que contém uma medida pré-definida que é comparada com os dados reais analisados em busca de desvios estatísticos significantes.

Finalmente, o terceiro e último tipo básico de classificação diz respeito à reação do IDS frente a um possível ataque. Sendo assim, um IDS pode ser definido como [DEBAR, 2004]:

- **Passivos**. Detectam uma possível violação da segurança, registram a informação e geram um alerta;
- **Reativos**. Projetados para responder a uma atividade ilegal, como por exemplo, enviando ao usuário do sistema uma medida de reprogramação do firewall para impedir o tráfego desta fonte hostil.

Para a segurança dos recursos de rede, acredita-se que o conhecimento dos tipos de IDS facilita a decisão sobre qual IDS melhor atende as necessidades da organização. A melhor opção de escolha deve aliar custos econômicos e propriedades desejadas, mantendo um alto nível de vantagens e um número controlado de desvantagens.

2.2.2 IDS Baseado em Agentes

Muitos sistemas IDS executam coleta e análise de dados de forma centralizada, empregando uma arquitetura monolítica [SILVA, 2006]. Os dados são coletados através de um simples *host*, seja a partir de escuta de rastros ou monitoramento de pacotes na rede, e analisados através de um simples módulo usando diferentes técnicas. Outros IDS executam a coleta de dados distribuída utilizando módulos distribuídos em *hosts* que são monitorados, mas continuam sendo recolhidos e encaminhados a um elemento central onde são analisados

através de um mecanismo monolítico [SILVA, 2006]. Geralmente, existem problemas em relação a essas arquiteturas:

- O analisador central é o ponto de falha;
- A escalabilidade é limitada;
- As reconfigurações ou adição de capacidades para o IDS tornam-se mais difíceis;
- A análise dos dados da rede pode conter erros.

A utilização de agentes de software no desenvolvimento de sistemas IDS [SILVA, 2006] proporciona a introdução das características das tecnologias de agentes no projeto desses sistemas. Um IDS baseado em agentes consiste em um conjunto de componentes autônomos trabalhando juntos de forma cooperativa. Uma vez que os ataques mudam dinamicamente, as assinaturas também mudam. Sendo assim, esses agentes têm a capacidade de aprender novas assinaturas ou detectar tráfego anormal originado por novos ataques.

O estudo de IDS baseado em agentes tem como objetivo sanar algumas deficiências contidas em arquiteturas distribuídas baseadas em Cliente-Servidor tais como [SILVA, 2006]:

- A existência de um analisador central que considerado como um ponto de falha;
- Tráfego excessivo sobre a rede (devido ao analisador central);
- Dificuldades em aplicar reconfigurações às estações com sensores de detecção, após uma modificação do sistema.

Dentre os principais projetos de Sistemas de Detecção de Intrusão baseados em agentes, pode-se citar o CAMNEP (*An Intrusion Detection System for High-Speed Networks*) [REHAK, 2008], CIDS (*Cougaar based Intrusion Detection System*) [DASGUPTA, 2005], MIDS (*Multi-Level and Multi-Agent Intrusion Detection System*) [BRIFFAUT, 2006] e o projeto objeto de estudo deste trabalho chamado IDS-NIDIA [LIMA, 2001].

2.3 MOM (*Message Oriented Middleware*)

O MOM (*Message Oriented Middleware*) [MENASCÉ, 2005] é definido como uma infra-estrutura de software Cliente/Servidor que cria uma camada entre as aplicações de alto nível e as plataformas onde estão instaladas, substituindo a comunicação direta entre aplicações por um sistema de troca de mensagens. Ao contrário do RPC (*Remote Procedure Call*) e do OOM (*Object Oriented Model*), o MOM é uma forma assíncrona de comunicação, ou seja, não há uma espera bloqueante do emissor para o receptor, durante a troca de

mensagens. Se o serviço de mensagens oferece persistência e confiabilidade, então o receptor não precisa estar ativo e executando quando a mensagem é enviada. As mensagens são geralmente não tipadas e sua estrutura interna é de responsabilidade da aplicação.

Tipicamente, uma implementação MOM fornece um conjunto de APIs (*Aplication Programing Interface*) capaz de funcionar com um número relativamente vasto de plataformas e redes que varia entre implementações. As APIs fornecem um nível de abstração capaz de aumentar a portabilidade, interoperabilidade e flexibilidade das aplicações que correm sobre a MOM. Usando as APIs os programadores são libertados dos detalhes das várias plataformas e protocolos, reduzindo assim a complexidade da implementação das comunicações das aplicações.

A MOM está presente em ambos os lados de uma arquitetura Cliente/Servidor e serve de ligação entre ambos, suportando chamadas assíncronas entre aplicações intervenientes. É possível adicionar sincronismo à arquitetura usando, por exemplo, RPC (*Remote Procedure Call*) em conjunto com MOM. Conseguindo abstrair da plataforma onde executa, uma aplicação pode trocar mensagens com outros programas residentes em plataformas diferentes.

Uma implementação MOM trata dois tipos de mensagens: as persistentes e as não persistentes. Uma mensagem importante pode ser marcada como persistente de forma a ser armazenada em memória não volátil, assim em caso de falha, os seus dados não são perdidos. As mensagens não persistentes são guardadas em memória volátil.

Os dados que podem ser transmitidos numa mensagem são:

- Dados formatados (variáveis strings);
- Pedidos de execução (chamadas a funções);
- Ou ambos.

As mensagens de um MOM são transmitidas por gestores de pilhas denominados MQM (*Message Queue Manager*) [MENASCÉ, 2005]. A finalidade de um gestor de pilhas é fornecer um ambiente para aplicações que usem as pilhas de mensagens ou para aplicações como a MQI (*Message Queue Inteface*). As suas funções são providenciar armazenamento confiável para as mensagens armazenadas nas pilhas, gerir acessos concorrentes a dados, assegurar segurança e autenticidade, e providenciar funções especiais de emparelhamento de mensagens. A Figura 2.3 ilustra a descrição da arquitetura interna de um MOM.

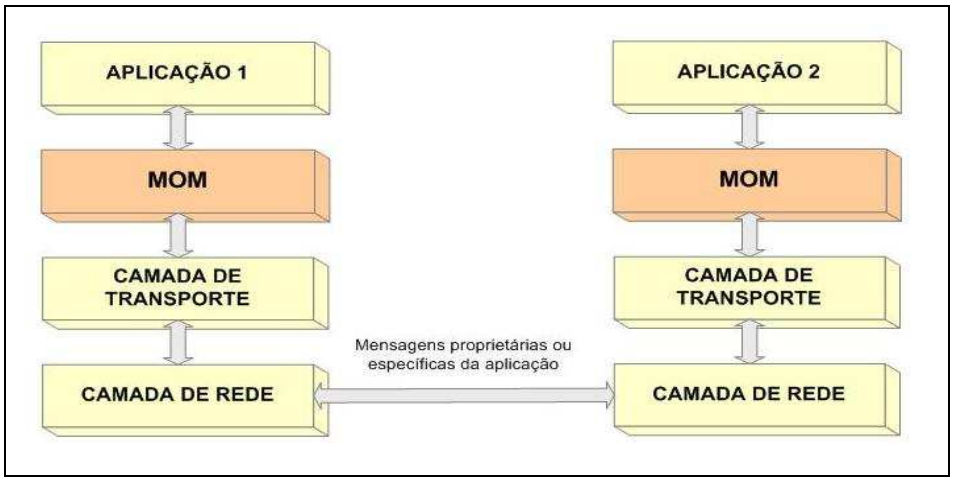

Figura 2.3 – Descrição geral da arquitetura MOM [MENASCÉ, 2005].

Nem sempre a aplicação que roda sobre MOM está disponível para receber mensagens, ou por estar processando mensagens anteriores ou por estar *offline*. Nestas situações, as mensagens são guardadas numa fila onde ficam à espera que a aplicação se torne disponível. Quando ocorre um evento, a aplicação cliente delega à MOM a função de notificar a aplicação servidora acerca desse evento. Por esse motivo, as implementações MOM são adequadas para ambientes sem fio (devido a possíveis falhas de conexão) e mais apropriadas para aplicações orientadas a eventos, embora também ofereçam um mecanismo conceitual para comunicação homóloga em sistemas orientados a objeto.

As vantagens do emprego da MOM nas comunicações entre aplicações são [MENASCÉ, 2005]:

- **Comunicação Assíncrona**. Permite independência temporal e execução não bloqueante, pois os processos não são obrigados a esperar uns pelos outros;
- **Usa uma aproximação *"send and forget"***. Desse modo o emissor não tem que esperar que o receptor receba a mensagem, aguardando somente que a mensagem seja transmitida e guardada na pilha de recepção do receptor;
- **Complexidade reduzida de programação**. Permite que ocorram alterações nos protocolos de rede, ou mesmo em arquiteturas onde correm as aplicações sem a necessidade de criar códigos específicos para a rede ou para a plataforma;
- **Introdução de clientes e servidores de forma transparente**. Pois não existe ligação direta entre as aplicações.

Entretanto, um dos grandes problemas desta arquitetura é o fato de não existir um padrão definido que regule as várias implementações. Assim, as implementações existentes

são proprietárias, o que por efeito torna diferentes implementações incompatíveis. Sendo obrigado a usar apenas um produto, o cliente fica dependente do fabricante da implementação escolhida.

No MOM tradicional, as mensagens são dirigidas aos seus destinatários, embora o emissor e o receptor estejam fracamente acoplados e não necessitam de sincronização para se comunicar. Um típico projeto desse modelo pode ser descrito nos sistemas *Publish-Subscribe*, onde as fontes publicam as mensagens para toda a rede e os interessados as recebem. Este modelo é representado na Figura 2.4.

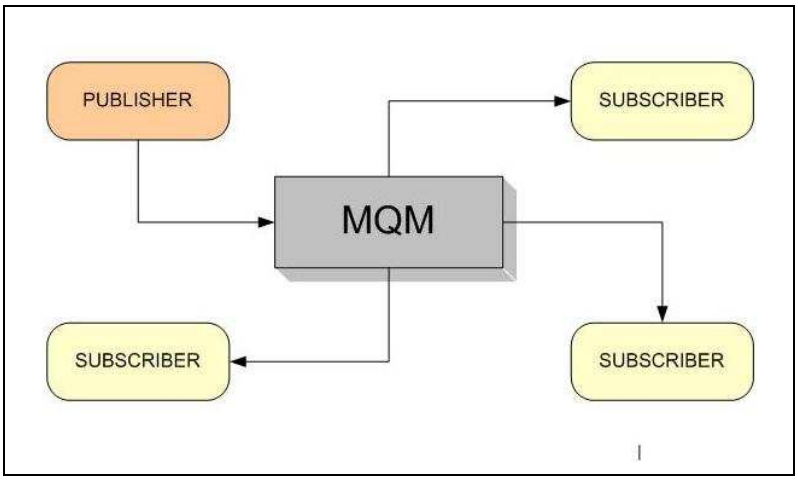

Figura 2.4 – Canal *Publish-Subscribe* [MENASCÉ, 2005].

Como exemplos de MOM, destaca-se o WebSphere MQ da IBM [IBM, 2008], o TIBCO Rendezvous [TIBCO, 2008] e o provedor de mensagens OpenJMS da Sun Microsystems [OPENJMS, 2008].

2.4 Autenticação

Uma autenticação compreende qualquer método que permite comprovar de maneira segura certas características de um objeto, como sua origem, sua identidade e sua integridade. [LUCENA, 2003]. Existem dois grandes grupos dentro dos métodos de autenticação existentes:

- **Autenticação de Mensagem**. Garante a procedência de uma mensagem, identificando a entidade que a emitiu. Por definição, este tipo de autenticação envolve também a integridade dos dados, ou seja, permite detectar se o conteúdo da mensagem foi alterado acidentalmente ou maliciosamente;

- **Autenticação de Usuário**. Garante que um usuário é quem afirma ser. Nesse processo se confirma a identidade do usuário em tempo real, isto é, no momento em que o verificador e o usuário estão se comunicando.

2.4.1 Autenticação de Mensagens

As técnicas mais utilizadas para prover autenticação de mensagens, segundo [MENESES, 1997], são: MAC (*Message Authentication Code*) e Assinatura Digital. Na autenticação MAC, os códigos são projetados especialmente para aplicações que requerem integridade dos dados (porém, não necessariamente sua privacidade). Um MAC é um tipo de função unidirecional que toma duas entradas (uma mensagem e um chave secreta) e retorna uma cadeia de tamanho fixo, sendo impossível obter a mesma saída sem conhecer a chave.

A Assinatura Digital possui outro meio de autenticação que simula uma assinatura real. Para isso, gera-se uma prova digital que somente o emissor da mensagem pode criar, mas que todos podem identificar como pertencente a esse emissor. Uma codificação utilizando a chave privada do emissor serve como assinatura, pois somente o proprietário dessa chave pode criá-lo e todos podem verificá-la com a chave pública correspondente. A assinatura pode aplicar-se a toda a mensagem ou a um pequeno bloco de dados que seja função da mensagem, como por exemplo, o valor de *hash*.

2.4.2 Autenticação de Usuário

As técnicas de autenticação de usuário se classificam em três categorias, segundo [MENESES, 1997], dependendo do que deve apresentar o usuário para demonstrar sua identidade:

- **Algo conhecido**. O usuário deve demonstrar que conhece algum tipo de informação secreta, tal como uma palavra chave, um número de identificação pessoal (PIN) ou uma chave privada;
- **Algo que possui**. O usuário requer, para identificar-se, algum tipo de dispositivo como um cartão magnético, um cartão inteligente, um gerador de senhas ou uma chave eletrônica;
- **Algo inerente (próprio do indivíduo)**. Nesse caso, alguma característica fisiológica do usuário (biometria) é requerida, por exemplo, impressão digital, voz,

retina ou íris. Esta categoria pode se considerar um caso particular da anterior, onde o dispositivo é o próprio usuário.

As técnicas de autenticação de usuário baseadas em algo conhecido, podem se classificar, por sua vez, em três categorias de acordo com o nível de segurança que oferecem [MENESES, 1997]:

- **Autenticação Débil**. É o tipo de autenticação mais difundido. Nesse caso, uma senha ou palavra chave (*password*), associada a cada usuário, é o segredo compartilhado entre o usuário e o sistema. Para identificar-se, o usuário entra com um identificador (*login*) e sua senha. Se o sistema comprovar que esta senha coincide com a armazenada pelo identificador, o acesso é liberado. As senhas são fixas e reutilizáveis, o que compromete a identidade do usuário caso elas sejam apreendidas pelo atacante;
- **Autenticação Forte**. Neste tipo de autenticação, a entidade verificada prova sua identidade frente à entidade verificadora, demonstrando que conhece um segredo associado com sua identidade, porém sem revelar este segredo ao verificador. Este procedimento se dá respondendo a um desafio variável. Os mecanismos de autenticação forte baseados em criptografia simétrica requerem que ambas a entidade verificadora e a verificada compartilhem uma chave secreta. Já os mecanismos de autenticação forte baseados em criptografia de chave assimétrica, a entidade verificada demonstra que possui sua chave privada, seja decifrando um desafio com sua chave privada ou assinando digitalmente o desafio;
- **Autenticação de Conhecimento Nulo**. Os protocolos de conhecimento nulo ZK (*zero-knowledge*) permitem uma entidade a ser verificada, demonstrar o conhecimento de um segredo sem revelar nenhuma informação útil para a entidade verificadora. Os sistemas de prova interativa são exemplos deste tipo de protocolo. Nesse caso, o objetivo da entidade verificada é convencer o verificador de que possui um segredo, respondendo corretamente as perguntas que requerem conhecimento desse segredo para serem respondidas. Estes protocolos empregam técnicas assimétricas, porém não utilizam assinatura digital ou criptografia com chave pública, cifras de blocos, números de seqüência e selo de tempo (*timestamp*).

2.4.3 Arquitetura de Autenticação

As técnicas de autenticação descritas anteriormente implicam algum tipo de chave ou segredo compartilhado entre as partes comunicantes. Deve existir, portanto, uma infra-estrutura de segurança que se encarregue da geração, distribuição e gerenciamento dos segredos. Esta infra-estrutura fornecerá uma base segura a toda organização e deverá ser acessível a todas as aplicações e objetos da organização que necessitem de segurança.

Em uma infra-estrutura de autenticação, uma terceira parte confiável (*Trusted Third Part* - TTP) ou Autoridade de Autenticação (*Authentication Authority* - AA) proporciona serviço de autenticação aos usuários. Esta autoridade governa um conjunto de recursos localizados em dispositivos que fazem parte de uma área de atuação denominada domínio. Para autenticar-se, o usuário compartilha uma credencial (na forma de um certificado) com a AA. Em seguida, a AA guarda a cópia desta credencial em uma base de dados segura. Dependendo do tipo de credencial, o usuário pode simplesmente gravá-la e armazená-la em algum meio físico (por exemplo, um cartão inteligente).

Durante um processo de autenticação, o usuário apresenta sua credencial (por exemplo, *login* e *password*) ou o resultado de uma operação criptográfica que envolve sua credencial a AA. Em seguida, a AA valida a credencial utilizando os dados guardados em sua base de dados. Se a credencial fornecida pelo usuário e a armazenada na base de dados coincidem, ou se o resultado de uma operação criptográfica sobre a credencial armazenada na base de dados é igual à informação fornecida pelo usuário, a identidade do usuário é considerada autêntica.

No entanto, o usuário necessita compartilhar uma credencial com cada domínio e se autenticar cada vez que quiser acessar algum recurso. Dada a diversidade de plataformas, sistemas operacionais e softwares de controle de acesso, o mais desejável é registrar-se em múltiplos sistemas de uma só vez, através de uma única transação. Nasce assim, o registro único chamado SSO (*Single Sign-On*). Com o SSO, usuário se autentica somente uma vez para acessar os múltiplos sistemas. O registro único pode ser usado em infra-estruturas com diversos AAs, isto é, implementadas em diferentes plataformas e governadas por diversas organizações.

As arquiteturas mais simples são as que usam um só conjunto de credenciais. As duas mais importantes são [ECHAVARRÍA, 2007]:

- **Sistemas baseado em Token (*Token-Based*)**. O usuário obtém um *token* temporário depois de autenticar-se com êxito junto a um TTP. Este *token* pode ser

guardado no dispositivo do usuário e reutilizado para provar sua identidade a um TTP de domínios de autenticação secundários. Para validar o *token* de um usuário, estas TTPs utilizam métodos criptográficos baseados nas chaves secretas estabelecidas previamente entre elas e a TTP de domínio de autenticação primário. Estas chaves criptográficas permitem estabelecer uma relação de confiança entre os diferentes domínios de autenticação;

- **Sistemas baseados em PKI (*PKI-Based*)**. O usuário se registra primeiramente em uma AA confiável denominada CA (*Certification Authority*). Durante o processo de registro, os usuários se identificam através de um conjunto de credenciais. O software do usuário gera um par de chaves assimétrico, e a chave pública é oferecida a CA para sua certificação. Após receber as credenciais do usuário e a chave pública, a CA verifica se as credenciais são válidas. Sendo válidas, esta gera um certificado de chave pública e retorna ao usuário. Este certificado e a chave privada são guardados de maneira segura no dispositivo do usuário ou em outro meio. Estes são usados para provar a identidade do usuário frente a outras CAs em solicitações de autenticação posteriores. Nesta arquitetura, a relação de confiança entre a CA primária e as CAs secundárias se estabelece através de um certificado expedido pela CA primária à CA secundária.

2.5 Autorização

A autorização está estreitamente ligada com a autenticação. Uma vez que o usuário validou sua identidade para acessar algum recurso, é necessário restringir suas ações de acordo com quem ele diz ser, o que está tentando fazer, etc. [STELL, 2005]. A autorização dota o usuário de privilégios para poder efetuar certas operações com dados ou recursos protegidos.

2.5.1 Modelos de Controle de Acesso

Um fator fundamental para determinar o funcionamento em torno da autorização é a definição do modelo de controle de acesso que se vai estabelecer. Os mecanismos utilizados para restringir o acesso aos recursos são geralmente uma, ou a combinação, das duas formas seguintes [ECHAVARRÍA, 2007]:

- **DAC (*Discretionary Acccess Control*)**. Este mecanismo deixa as decisões de controle de acesso a cargo do proprietário do recurso, de maneira que ele decide quem pode realizar determinadas ações sobre os recursos possuídos. Assim, se um usuário tem permissões de acesso a um determinado recurso, então ele pode disponibilizar este mesmo recurso indiretamente a outro usuário.
- **MAC (*Mandatory Access Control*)**. Este controle de acesso se baseia nas regras estabelecidas por uma autoridade central. MAC restringe o acesso aos objetos baseando-se na sensibilidade da informação que contém e na autorização formal dos usuários para ter acesso a esta informação. Os objetos são considerados como entidades passivas que armazenam informações, enquanto os usuários são entidades ativas que realizam pedidos de acesso aos objetos.

Existem diversos modelos de controle de acesso. Os mais comuns são [ECHAVARRÍA, 2007]:

- **Controle de Acesso baseado em Identidade (IBAC)**. Nesse caso, as permissões de acesso a um recurso estão diretamente associadas ao identificador do usuário (nome do usuário). Portanto, o acesso ao recurso só é garantido quando existe esta associação. Um exemplo de IBAC são as Listas de Controle de Acesso (ACL), encontradas comumente em sistemas operacionais e serviços de segurança de rede. Uma ACL contém os identificadores dos usuários junto com seus direitos de acesso a um determinado recurso, como ler, escrever, executar, etc. Esta estrutura básica de autorização estende unicamente o conceito de autenticação, já que o usuário não pode autenticar-se corretamente junto a um guardião do recurso, se sua solicitação de acesso é negada;
- **Controle de Acesso baseado em Regras (RBAC)**. Restringe o acesso aos recursos baseando-se na função ou papel que desempenha o sujeito dentro da organização. As permissões para acessar um recurso são atribuídas a cada papel, ao invés de associá-las diretamente ao identificador do sujeito. O RBAC é escalável e reduz significativamente a quantidade de informações de administração necessária, já que as permissões não são assinadas constante e individualmente aos usuários;
- **Controle de Acesso baseado em Atributos (ABAC)**. Neste controle de acesso, os privilégios são estabelecidos com base no conjunto de atributos que possui o

usuário e uma política que os determina. ABAC é a convergência natural dos modelos de controle de acesso IBAC e RBAC. A representação das políticas em ABAC são semanticamente mais ricas e expressivas. Além disso, possuem uma maior granularidade, pois podem se basear em qualquer combinação de atributos de sujeito, de recursos e de ambiente;

- **Controle de Acesso baseado em Etiquetas (LBAC)**. Nesse caso, um conjunto totalmente ordenado de etiquetas de segurança se combina a um grupo de categorias e formam uma malha. Com ela se obtém um conjunto de classes de segurança que varia desde a mais baixa até a mais alta. Geralmente, o modelo LBAC é utilizável quando existe um número relativamente pequeno de etiquetas de segurança e categorias, pois só tem efeito para certos cenários de segurança pouco granulados e carentes de flexibilidade e escalabilidade.

2.6 IDS-NIDIA

A proposta do sistema IDS-NIDIA [LIMA, 2001] [OLIVEIRA, 2006] [SILVA, 2006] é apresentar um modelo de um sistema de detecção de intrusão, em tempo real, baseado na noção de sociedade de agentes inteligentes capazes de detectar novos ataques através de uma rede neural. Geralmente, este modelo leva em conta as características do recurso e não obtém nenhuma informação relevante de segurança do ambiente.

O IDS-NIDIA é inspirado no modelo lógico do CIDF [STANIFORD-CHEN, 1998], possuindo para este fim, agentes com função de geradores de eventos (agentes sensores), mecanismos de análise dos dados (agentes de monitoramento e de avaliação de segurança), mecanismos de armazenamento de histórico (base de dados) e um módulo para realização de contramedidas (agente controlador de ações). Além disso, existem agentes responsáveis pela integridade do sistema e pela coordenação das atividades do IDS.

Os objetivos gerais do projeto IDS-NIDIA são:

- Gerar índices de suspeita de ataque por meio de análise das informações coletadas de *logs* de *hosts* e de pacotes de tráfego de rede;
- Tomar contramedidas baseados nos índices obtidos;
- Aprender com as informações obtidas atualizando suas bases de conhecimento.

O modelo proposto [LIMA, 2001] prevê a metodologia de detecção por abuso e anomalia para garantir uma robustez maior do sistema. Essa escolha se deu em virtude da

maioria dos ataques poderem ser codificados, de maneira a capturar e registrar variantes das atividades que exploram as mesmas vulnerabilidades.

2.6.1 Arquitetura do IDS-NIDIA

A arquitetura do IDS-NIDIA é dividida em seis camadas e cada camada possui atividades a desempenhar, sendo que estas atividades são executadas através do comportamento dos agentes que a compõe. É através desses agentes também, que as camadas se comunicam trocando informações importantes para o desempenho das atividades. A Figura 2.5 mostra a arquitetura do IDS-NIDIA [SILVA, 2006].

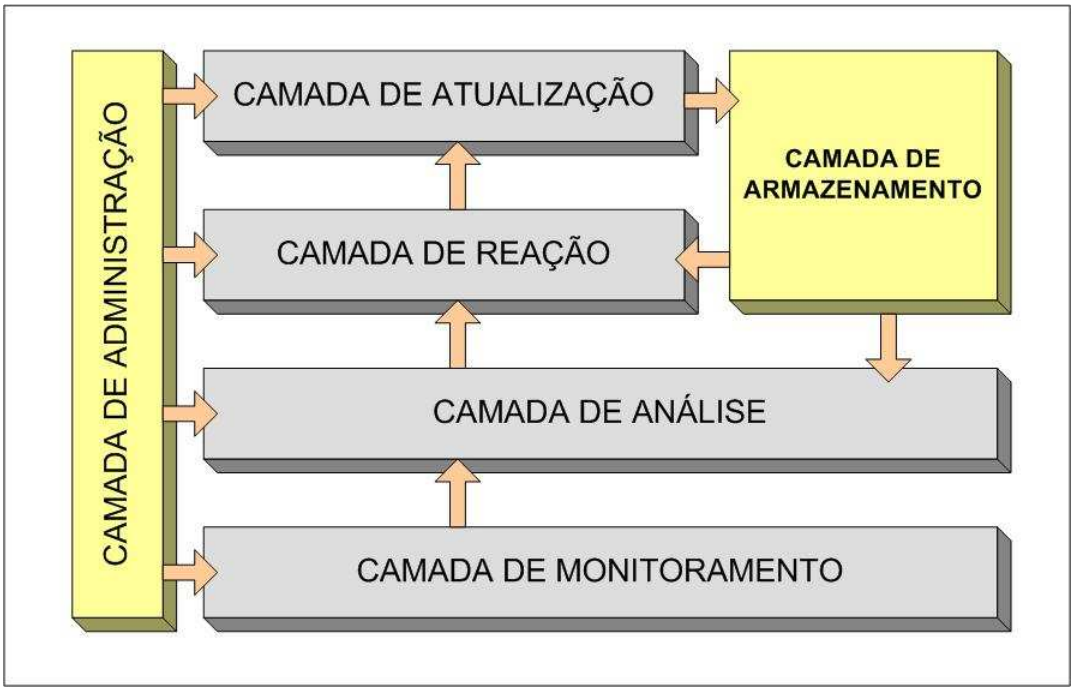

Figura 2.5 – Arquitetura do IDS-NIDIA.

A Camada de Monitoramento é responsável por capturar a ocorrência de evento no meio exterior e fornecer dados do mesmo para o resto do sistema. Nesta camada, os Agentes de Monitoramento do Sistema (SMA) estão localizados, sendo que eles se dividem em duas categorias: Os Agentes Sensores de Rede (NSA), que capturam os pacotes que estão trafegando na rede; e os Agentes Sensores de Host (HSA), que trabalham coletando informações de um host e disponibilizando-as para analise.

A Camada de Análise tem a função de analisar os eventos recebidos da Camada de Monitoramento. Nesta camada, os eventos coletados são formatados de maneira que padrões

de ataque possam ser identificados e confirmados como um ataque verdadeiro. Aqui são utilizadas as seguintes bases de dados de conhecimento que serviram de análise: A base de padrões de intrusão (IIDB), a base de incidentes de intrusão (DFDB) e a base de estratégias (STDB). Nesta camada, os Agentes de Avaliação do Sistema (SEA) analisam e emitem um grau de suspeita sobre os eventos que foram previamente formatados.

A Camada de Reação é responsável por tomar contramedidas, caso um incidente de segurança seja confirmado. Nesta camada, os Agentes Controladores do Sistema (SCA) executam as contramedidas de acordo com as bases de dados de estratégia (STDB) e de ações (RADB).

A Camada de Atualização é responsável por manter todas as bases de dados do sistema sempre atualizadas. Nesta camada, os Agentes de Atualização do Sistema (SUA) têm a responsabilidade de manter a integridade e consistência das informações armazenadas, sendo os únicos capazes de alterar essas bases de dados.

A Camada de Administração é composta pelos Agentes Controladores Principais (MCA), que são responsáveis pela administração e integridade de todos os agentes do sistema.

A Camada de Armazenamento mantém de forma persistente as informações oriundas das demais camadas. Aqui estão contidas todas as bases de dados utilizadas pelo IDS-NIDIA.

2.6.2 Operações do IDS-NIDIA

O funcionamento do IDS-NIDIA se dá através das interações entre os agentes que compõe suas camadas. Cada camada possui um objetivo bem definido e cada objetivo é alcançado através da interação e troca de informações entre as camadas. Portanto, o IDS-NIDIA possui três fases que define o seu funcionamento: a captura, a análise e as contramedidas.

Na fase de captura, atuam os agentes HSA e os agentes NSA, onde eles têm a responsabilidade de capturar a ocorrência de eventos e fornecer informações sobre os mesmos para o resto do sistema. O agente HSA faz a leitura dos *logs* de segurança de servidores específicos, e repassa os dados para o agente SMA para serem formatados. Posteriormente, tais informações pré-processadas são enviadas deste agente SMA para um agente SEA, para que sejam avaliadas. O processo de captura é demonstrado na Figura 2.6.

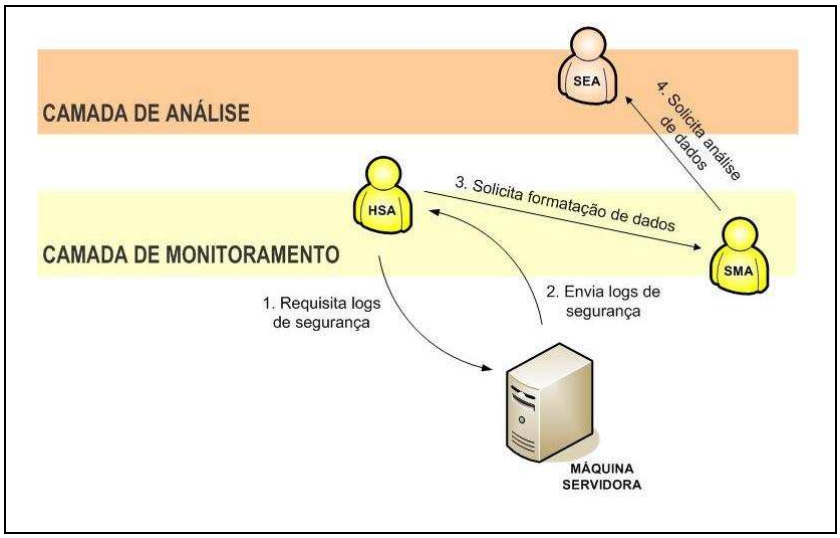

Figura 2.6 – Funcionamento da Captura do IDS-NIDIA.

Na fase de análise, atua os agentes SEA que são especializados o tratamento do cabeçalho dos pacotes e na inspeção do conteúdo das conexões TCP. O produto gerado por esses agentes é o grau de suspeita, que é enviado para os agentes. O processo de análise é demonstrado na Figura 2.7.

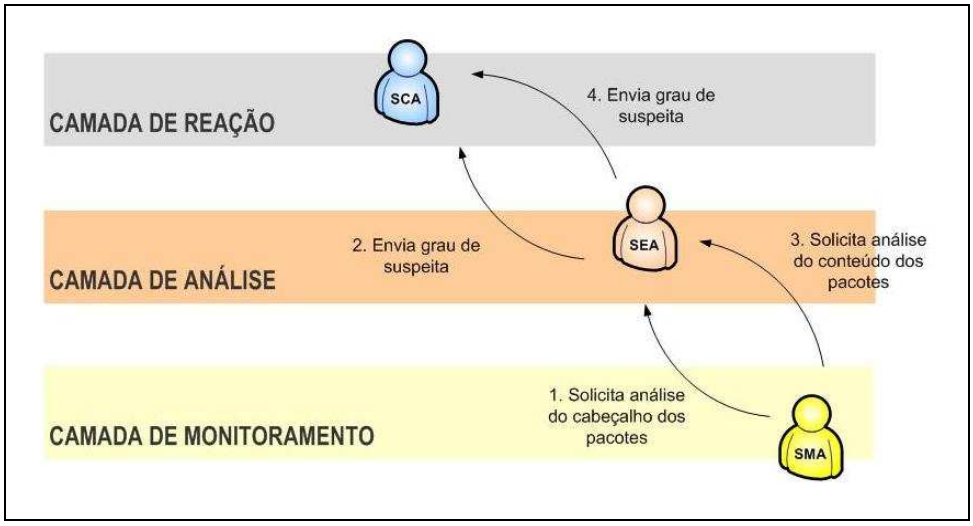

Figura 2.7 – Funcionamento da Análise do IDS-NIDIA.

Na fase de contramedidas, são executadas as medidas cabíveis, baseadas no grau de suspeita do ataque. O agente SCA notifica um ataque ao Agente de Segurança Local (LSIA), que são responsáveis pela verificação do estado do IDS como um todo. Em seguida, ele solicita ao agente SUA, atualização no DFDB para registro das informações sobre o novo ataque. O processo de tomada de contramedidas é demonstrado na Figura 2.8.

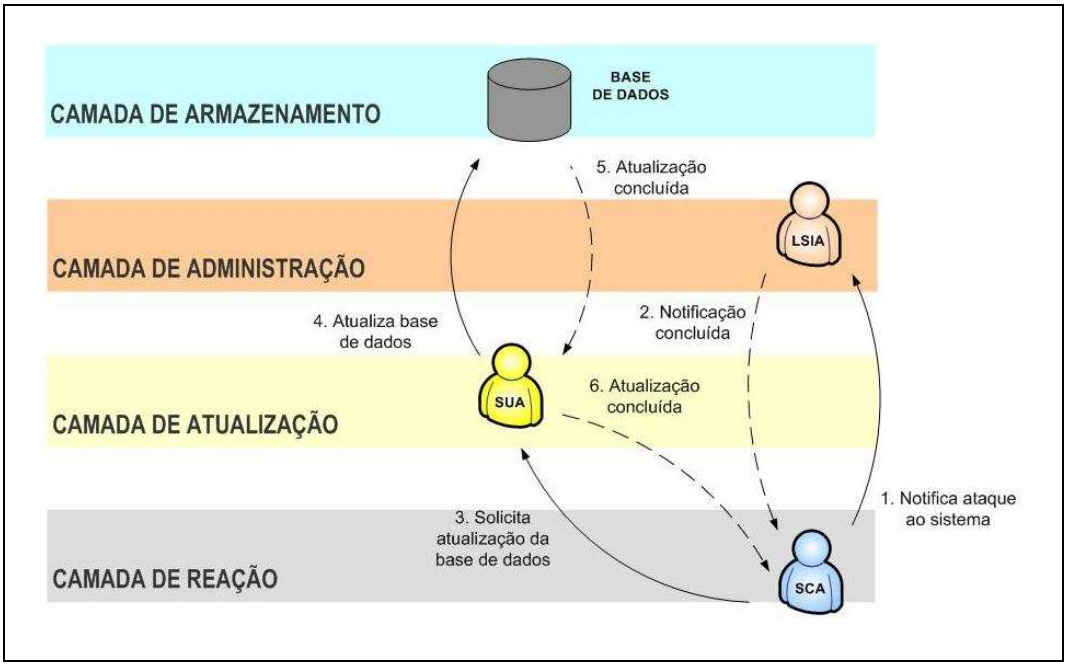

Figura 2.8 – Funcionamento da Contramedida do IDS-NIDIA.

2.7 Conclusão

Neste capítulo, apresentou-se as fontes de investigação que foram úteis para a compreensão do problema e que serviram de base para a construção da solução proposta por esta dissertação. Nesse sentido, a importância foi dada ao estudo dos conceitos relacionados a agentes e sistemas multiagentes, sistemas de detecção de intrusão (e suas recentes pesquisas utilizando agentes), *middlewares* orientado a mensagens, autenticação e autorização e ao projeto IDS-NIDIA.

Na seção de Agentes e Sistemas Multiagentes, a definição de agente, suas principais características, aplicabilidades e áreas de investigação que atuam, bem como, o conceito de sistemas multiagentes e a ferramenta utilizada para o seu desenvolvimento foram apresentados. Na seção de IDS, apresentou-se uma visão geral do sistema envolvendo sua definição, características, classificação de acordo com critérios de detecção ou tipo de reação, e suas recentes pesquisas envolvendo o uso de agentes na sua estrutura. Na seção MOM, apresentou-se uma descrição geral de sua infra-estrutura, bem como, as vantagens e desvantagens quanto ao uso de sua arquitetura. Na seção de Autenticação e Autorização, a definição, os métodos e a arquitetura de autenticação, assim como, a definição e os modelos de controle de acesso, foram apresentados. Finalmente na seção IDS-NIDIA, apresentou-se a proposta do sistema, sua arquitetura e o seu funcionamento.

3 Solução Proposta

Este capítulo dedica-se a uma visão geral dos mecanismos desenvolvidos para integrar a solução proposta nesta dissertação. Dessa forma, pretende-se apresentar as funcionalidades de cada mecanismo e o papel que eles desempenham dentro do contexto da segurança e da confiabilidade relacionados ao funcionamento dos agentes. Neste capítulo, pretende-se também mostrar como a solução pode fornecer proteção quando incorporada à execução e comunicação dos agentes entre as diferentes camadas do IDS-NIDIA.

O modelo proposto é constituído de quatro partes: proteção de informações de configuração, autenticação e autorização, gerenciamento de chaves e *timestamps*, e persistência de mensagens.

3.1 Proteção de Informação de Configuração

Em sistemas IDS executado por agentes, a comunicação pode ser realizada por meio de mensagens criptografadas. Entretanto, para que isso ocorra, todos os agentes que compõem o sistema necessitam antes passar pelo processo de registro de suas chaves públicas junto a um repositório de chaves como o XKMS Server. O modelo proposto para esse sistema contém medidas de segurança específicas para o registro dessas chaves no XKMS Server.

Uma dessas medidas de segurança é o uso de uma chave compartilhada e temporária denominada chave secreta, que é utilizada para criptografar e descriptografar as mensagens de registro entre os agentes e o XKMS Server. Essa chave também é única, pois é diferente para cada agente que efetua esse registro.

O processo de registro se inicia quando um agente entra em acordo com o XKMS Server e ambos geram uma mesma chave secreta, utilizando o algoritmo *Diffie-Hellman* [DIFFIE-HELLMAN, 2007]. Após isso, o agente inicia suas atividades e gera separadamente seu par de chaves público e privado. A chave pública gerada é criptografada com a chave secreta e o resultado é uma mensagem cifrada que é enviada para o XKMS Server. Em posse da mensagem cifrada, o XKMS Server se encarrega de descriptografá-la (utilizando a chave secreta) e armazenar o conteúdo (chave pública do agente) na sua base de dados PKI. Em seguida, o XKMS Server envia uma resposta ao agente contendo a sua chave pública criptografada com a mesma chave secreta, que será armazenada pelo agente utilizando o mesmo procedimento de decifragem descrito para o XKMS Server. Concluída a troca de

chaves públicas entre o agente e o servidor, a chave secreta é descartada e o registro é finalizado. O processo é mostrado na Figura 3.1.

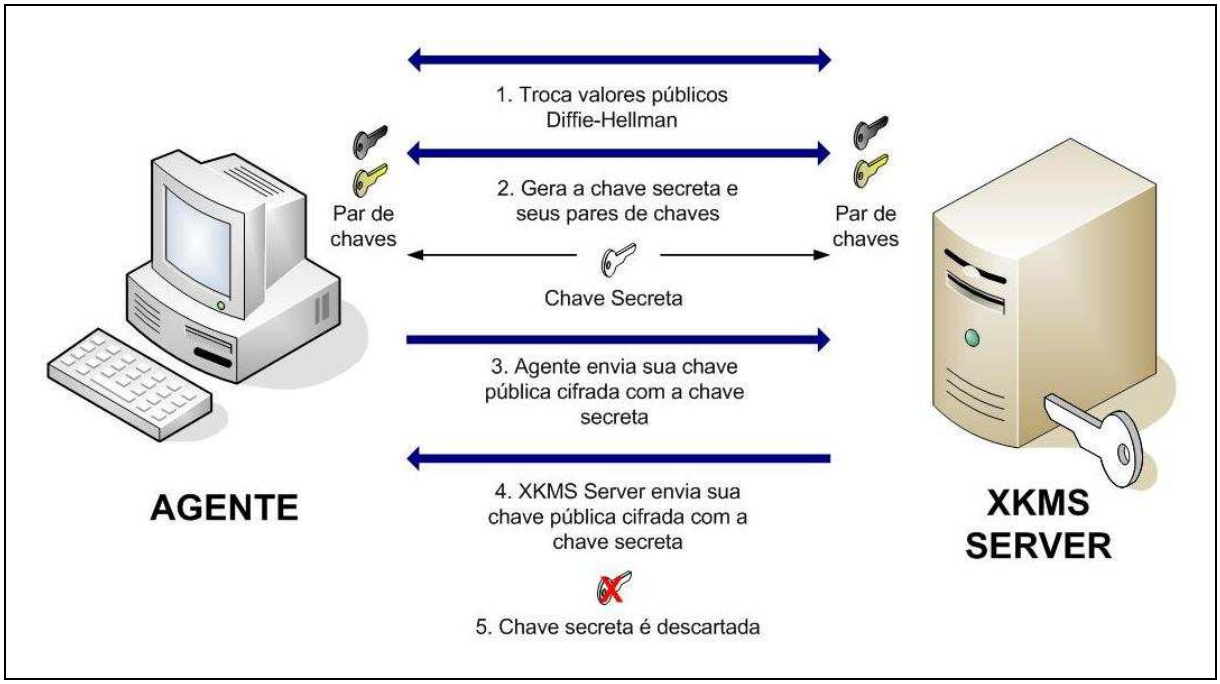

Figura 3.1 – Esquema de registro da chave pública pelo XKMS Server.

Contudo, o processo de registro de chaves no sistema adota um mecanismo de proteção que utiliza informações sensíveis do servidor, que são armazenadas e compartilhadas entre os agentes do sistema em arquivo local. Isto significa que os agentes necessitam obter informações de configuração do XKMS Server antes de iniciar qualquer atividade, e o servidor as disponibiliza através de um documento XML. Como essas informações são armazenadas em formato de texto legível, elas devem ser submetidas a um esquema de proteção, pois um intruso pode perfeitamente comprometer o sistema, se este direcionar seu ataque a essas informações sensíveis. Embora este procedimento não impeça totalmente a invasão do sistema, se estas informações de configuração puderem ser protegidas, o sistema será capaz de restringir consideravelmente atividades de intrusão, frustrando ataques dessa natureza.

Portanto, a outra medida de segurança proposta neste trabalho é o uso de criptografia para proteger as informações de configuração, que incluem os dados sobre o XKMS Server e sobre os parâmetros do algoritmo *Diffie-Hellman* que são lidas pelos agentes e pelo XKMS Server. Essas informações são armazenadas em arquivo, pois evita o ataque do Homem-do-Meio (*Man-In-The-Middle Attack*) [VAN TILBORG, 2005] e impede que o intruso interfira

na comunicação e consiga iludir tanto o XKMS Server quanto qualquer agente que esteja registrando ou localizando sua chave pública. Como os parâmetros contidos no arquivo não são trocados através da rede e o fato dos agentes possuírem entradas estáticas para o endereço MAC do XKMS Server, as informações de configuração armazenadas em arquivo representam uma dificuldade a mais para possíveis ataques. A Listagem 3.1 apresenta as informações contidas no arquivo de configuração, que são utilizadas como parâmetros iniciais para a comunicação segura entre os agentes do IDS-NIDIA.

Listagem 3.1 – Representação do arquivo de configuração.

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <configuration>
3    <xkms_server>
4      <ip_address>192.168.4.124</ip_address>
5      <mac_address>00-40-f4-a2-b9-6d</mac_address>
6    </xkms_server>
7    <diffie-hellman>
8      <prime_modulus_P>10895399[...]</prime_modulus_P>
9      <base_generator_G>9847766[...]</base_generator_G>
10     <bit_size_exponent_L>1024</bit_size_exponent_L>
11   </diffie-hellman>
12   <platform>win</platform>
13 </configuration>
```

Como dito anteriormente, o arquivo de configuração é um documento XML estático e legível que armazenam dados que podem comprometer seriamente o sistema caso ele seja detectado e tenha seu conteúdo analisado por intrusos. Neste arquivo, dados de identificação do XKMS Server na rede, como os endereços IP e MAC (linhas 4 e 5), e os parâmetros utilizados na geração da chave secreta (linhas 8 a 10), através do algoritmo *Diffie-Hellman*, estão contidos.

Partindo de uma visão mais detalhada, a solução propõe um esquema de proteção baseado na geração dinâmica das informações de configuração e seu armazenamento criptografado em arquivo. Este procedimento é realizado sempre que o XKMS Server é inicializado no sistema, e sua execução atenderá os seguintes passos:

- Um par de chaves público e privado é criado especificamente para esta solução, denominado “chaves de configuração” do XKMS Server;
- O arquivo é gerado em tempo de execução e seu conteúdo é criptografado utilizando a chave privada de configuração que, em seguida, é compartilhado para os agentes como um documento XML criptografado.

O procedimento de leitura desse arquivo é feito pelos agentes do sistema, porém é executado mediante a chave pública de configuração fornecida pelo XKMS Server. Esta chave pública somente é disponibilizada aos agentes devidamente autenticados e autorizados pelo XKMS Server, ou seja, apenas os agentes pertencentes ao sistema IDS têm acesso a essa chave. O documento XML criptografado, contendo as informações de configuração, é mostrado na Listagem 3.2.

Listagem 3.2 – Representação do arquivo de configuração criptografado.

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <xenc:EncryptedData xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
   Type="http://www.w3.org/2001/04/xmlenc#Element">
3    <xenc:EncryptionMethod
   Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
4     <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
5       <xenc:EncryptedKey>
6         <xenc:EncryptionMethod
   Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
7           <xenc:CipherData>
8             <xenc:CipherValue>Hr3Fj4GFn[...]</xenc:CipherValue>
9           </xenc:CipherData>
10      </xenc:EncryptedKey>
11    </ds:KeyInfo>
12    <xenc:CipherData>
13      <xenc:CipherValue>uu4iFAaIx[...]</xenc:CipherValue>
14    </xenc:CipherData>
15 </xenc:EncryptedData>
```

Como as etapas de geração, cifragem e armazenamento das informações de configuração são executados somente quando o XKMS Server é iniciado, isso garante uma

segurança a mais para o sistema, pois a cada inicialização do servidor, um par de chaves de configuração diferente é gerado e um arquivo XML cifrado diferente é criado.

3.2 Autenticação e Autorização

Como dito anteriormente, um bom projeto de sistemas IDS baseados em agentes deve incorporar meios de assegurar a execução e a comunicação dos seus componentes internos (agentes). Por esse motivo, torna-se necessário garantir a segurança das informações trocadas entre os agentes, através do reconhecimento de forma confiável da identidade das entidades envolvidas na comunicação e assegurar o controle eficaz do nível de acesso dessas entidades.

Quando se trata de segurança da informação, é fundamental garantir confidencialidade, integridade e disponibilidade. O controle de acesso, em particular, é um mecanismo fundamental para evitar operações não autorizadas no sistema. Sendo aplicado a um sistema de detecção executado por agentes, a função do controle de acesso permite garantir que apenas agentes autorizados possam executar operações e acessar recursos do IDS. Para isso, é necessário identificar e autenticar o agente que está solicitando o acesso aos recursos do IDS. Na identificação, deseja-se saber quem está tentando operar o IDS. Na autenticação, busca-se a garantia de que o agente é quem declara ser e que o mesmo faz parte do IDS.

Baseado nisso, a solução proposta integra mecanismos robustos para autenticação e autorização dos agentes para garantir segurança na troca de mensagens. O objetivo é permitir que somente os agentes do sistema tenham acesso às funções do XKMS Server, necessárias para efetuar uma comunicação segura. Através do mecanismo de autenticação, o XKMS Server é capaz de restringir o acesso a serviços de registro e localização de chaves públicas, disponibilizando-os somente aos componentes que integram a arquitetura do sistema. Com a utilização do mecanismo de autorização, o XKMS Server terá controle absoluto sobre quais recursos o agente devidamente autenticado terá acesso, seja operação de registro, localização ou ambos.

A solução propõe um modelo de comunicação entre os agentes e o XKMS Server de forma que suas identidades (tanto os agentes quanto o servidor) sejam estabelecidas de forma confiável. Para um processo de registro de chaves, por exemplo, o modelo apresenta os seguintes passos iniciais, conforme é ilustrado na Figura 3.2:

- O XKMS Server estabelece um canal seguro de comunicação utilizando a tecnologia de segurança denominada SSL (*Secure Socket Layer*), onde ocorrerá o

tráfego das informações de *login* dos agentes e o envio da chave pública de acesso ao arquivo de configuração. A utilização do protocolo SSL, permite criar comunicações seguras em ambientes onde ocorrem transações que envolvem dados confidenciais, encriptando toda a transmissão entre o cliente e o servidor. Essa etapa constitui o primeiro obstáculo de segurança que obriga a autenticação do agente;

- O agente fornece o seu "*login*", que consiste na entrada de um *username* (nome de usuário) e um *password* (senha), provendo ao servidor sua identificação. Essa identificação é um processo que possibilita aplicar todas as medidas básicas de segurança estipuladas para cada agente, como por exemplo, restringir o acesso do agente somente aos recursos pré-definidos pelo servidor. Essa etapa define o segundo obstáculo de segurança que obriga a autenticação do agente;
- O XKMS Server analisa as informações secretas contidas no *login* do agente e, confirmado sua autenticidade, este disponibiliza a chave pública para leitura das informações de configuração do XKMS Server codificadas em arquivo, que são necessárias para a geração da chave secreta;
- A chave secreta é gerada e compartilhada entre o agente e o XKMS Server e, em seguida, o processo de registro de chave pública é iniciado.

O SSL é um protocolo de comunicação que implementa um duto seguro para aplicações na Internet, de forma transparente e independente da plataforma [APACHE, 2008]. Sua proposta é permitir a autenticação de servidores, encriptação de dados, integridade de mensagens e, como opção, a autenticação do cliente, operando nas comunicações de aplicativos de forma interoperável. Este protocolo foi desenvolvido inicialmente pela Natscape Communications [NATSCAPE, 2008] tendo a sua especificação submetida ao grupo de trabalho W3C [W3C, 2008]. A solução utiliza esse protocolo na sua versão 3.0 (SSLv3), que tem as seguintes melhorias em relação às versões anteriores:

- Diminuição no número de rodadas de negociação;
- Escolha das cifras e compreensão por parte do servidor;
- Um suporte mais completo para a troca de chaves de algoritmos de cifragem;
- Possibilidade de renegociação das cifras em uso;
- Separação das chaves de autenticação e encriptação.

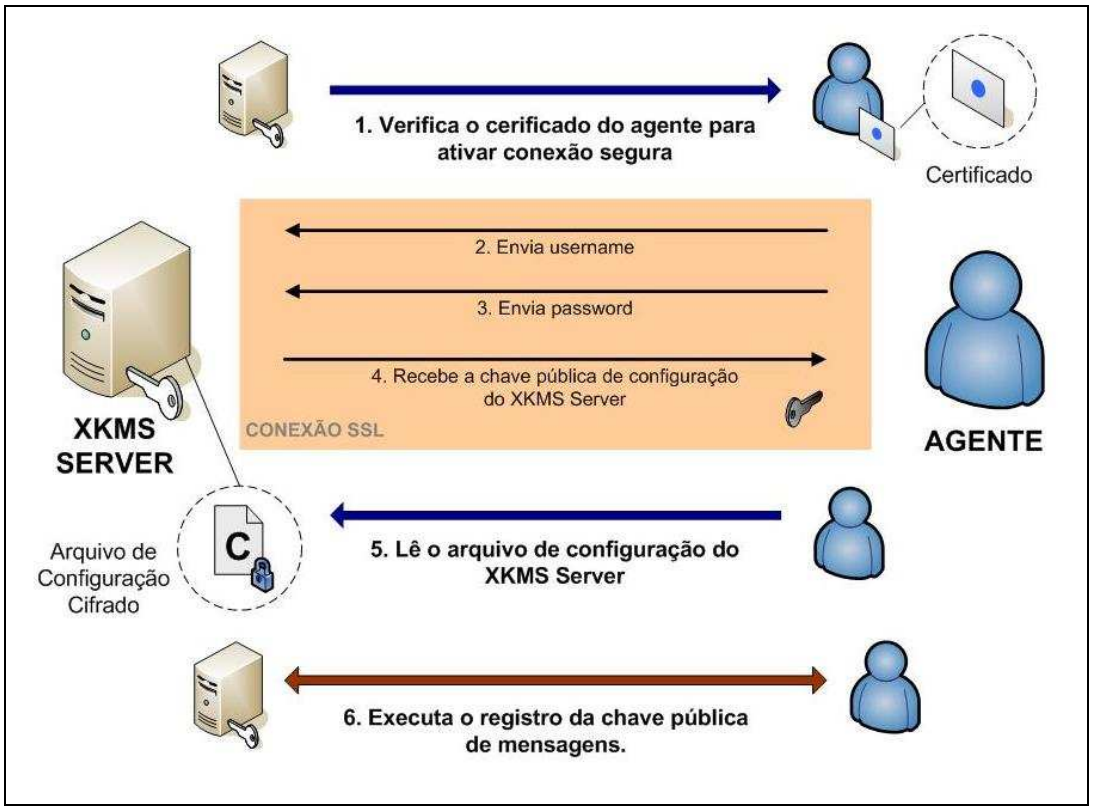

Figura 3.2 – Esquema de autenticação e controle de acesso ao registro de chaves.

O SSL atua entra as camadas de transporte TCP (*Transport Control Protocol*) e aplicação, sendo independente do protocolo de alto nível podendo rodar sob HTTP (*HyperText Transfer Protocol*), TelNet, FTP (*File Transfer Protocol*) e outros, de forma transparente.

Durante o processo de geração do canal seguro de comunicação (via SSL), os agentes fornecem suas credenciais, na forma de certificados, para autenticar-se junto ao XKMS Server. Este procedimento garante que o agente que esteja utilizando o certificado refere-se ao mesmo agente para o qual este certificado foi emitido. O XKMS Server detém o controle de todos os certificados confiáveis emitidos para os agentes do IDS, sendo que ele armazena essas informações dentro de um repositório específico. Sendo assim, a autenticação do XKMS Server só pode ser confirmada se o certificado fizer parte dos certificados confiáveis pré-configurados no servidor.

Quando as informações de *login*, fornecidas pelo agente, são confirmadas pelo XKMS Server como sendo de agentes pertencentes ao IDS, este se transforma em um agente válido

---

[1]principal é o termo para designar agentes que acessam informações e recursos mantidos num sistema.

de acesso, ou seja, um *principal*[1] do sistema. Através do *principal*, o XKMS Server controla quais operações (registro de chaves, localização ou ambos) são permitidas para o agente durante a comunicação.

A solução que prover mecanismos de autenticação e autorização é dividida em dois módulos, conforme é ilustrado na Figura 3.3:

- **Módulo *ConnectSSL***. Sua função é restrita a geração do canal seguro para envio das informações de *login* dos agentes. Este módulo é parte integrante do Módulo *SecurityJAAS*.
- **Módulo *SecurityJAAS***. Abrange todo o mecanismo, sendo responsável pelos serviços de autenticação e controle de acesso dos agentes propriamente ditos.

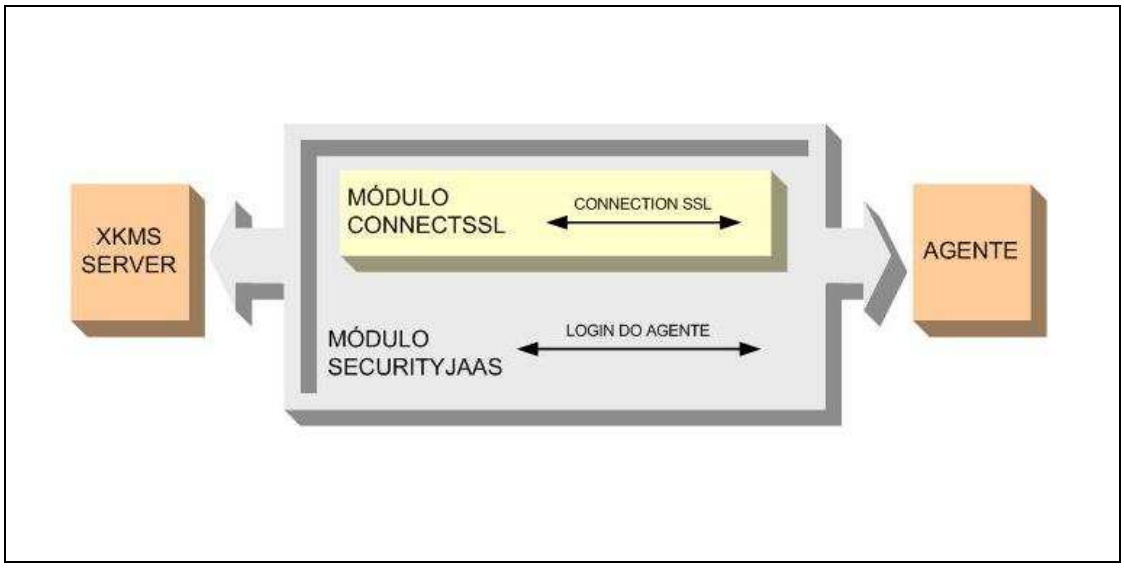

Figura 3.3 – Módulos do mecanismo de autenticação e autorização da solução proposta.

3.3 Gerenciamento de Chaves e *Timestamps*

Os administradores de sistemas de informática sempre tiveram um interesse especial em controlar o acesso às máquinas e recursos para impedir seu uso indevido e garantir a proteção das informações que nelas estão contidas. Quando estas máquinas e sistemas se encontram interconectadas através de uma rede, esse controle e essa proteção se tornam mais necessários e mais difíceis de prover, partindo de métodos convencionais.

A simples existência de uma rede local estendida ao longo de um edifício, por exemplo, possui um notável conjunto de riscos para a segurança da informação que por ela circula ou que estão armazenadas nos computadores a ela conectados. Quando estes sistemas

e redes locais estão, por sua vez, conectados a redes de âmbito nacional ou internacional, o problema torna-se mais delicado.

Para suprir esta deficiência, os organismos de normalização propuseram uma série de soluções para evitar possíveis ataques e operações ilegais a que estão submetidas às redes abertas. Estas soluções consistem em dotar as redes de uma série de serviços de segurança que utilizam em sua maioria técnicas criptográficas como ferramentas básicas.

A criptografia é considerada um componente crítico de qualquer sistema de segurança. Sendo assim, ela necessita ser corretamente implantada para evitar o acesso indevido às chaves que constituem a base do processo de encriptação. Na criptografia de chave pública, a manipulação de chaves é considerada um ponto fraco que deve ser levado em conta, pois um intruso poderia naturalmente manipular os processos com as chaves, ou falsificar a chave pública, disponibilizando-a aos usuários como se fosse autêntica. Por conta disso, os controles criptográficos são muito importantes para a implementação das políticas de segurança de um sistema, porém o seu uso introduz algumas preocupações quanto ao armazenamento e à distribuição das chaves criptográficas. Técnicas seguras de gerenciamento de chaves precisam ser utilizadas visando amenizar essas preocupações.

Na Criptografia [LUCENA, 2003], o gerenciamento de chaves está diretamente relacionado às medidas feitas para definir um bom projeto de um sistema criptográfico. Proteger os dados é importante, entretanto garantir medidas de segurança das chaves é fundamental para o sucesso de um sistema. Dentre essas medidas, pode-se citar a geração, troca, armazenamento, proteção, utilização, verificação e renovação de chaves.

Em um sistema IDS executado por agentes, uma ferramenta robusta de gerenciamento de chaves torna-se o elemento principal na segurança do sistema. O sistema IDS-NIDIA, por exemplo, dispõe de uma ferramenta de gerenciamento de chaves através de um XKMS Server, que tem papel importante na comunicação dos agentes. Essa ferramenta é baseada em um modelo similar ao esquema PKI (*Public Key Infrastructure*) desenvolvida a partir da adaptação da especificação de segurança XKMS (*XML Key Management Specification*) [OLIVEIRA, 2006]. A especificação XKMS permite fácil gerenciamento da PKI abstraindo a complexidade dessa infra-estrutura entre aplicações clientes e as entidades TTP (*Trusted Third Party*). Seus principais objetivos são:

- Criar uma camada abstrata entre a aplicação e a solução PKI, permitindo que diferentes soluções PKI sejam integradas a aplicação sem requerer qualquer modificação da mesma;

- Eliminar a necessidade de compreender a complexidade da sintaxe e semântica PKI, por parte da aplicação, fornecendo um simples protocolo baseado em XML para processamento das informações de chaves através de um serviço XKMS.

O papel do XKMS Server é receber várias requisições de chaves e retornar as respostas apropriadas, isto é, após um agente gerar seu par de chaves, ele pode registrar sua chave pública junto ao XKMS Server para habilitar outros agentes a localizarem e a utilizarem a chave pública registrada [OLIVEIRA, 2006].

Para efetuar um registro de chave, o agente envia sua chave pública e aguarda uma resposta do servidor. O XKMS Server recebe a chave do agente, armazena a informação na sua base de dados PKI e retorna uma mensagem de sucesso contendo sua chave pública anexada. Durante esse processo, todas as mensagens trocadas entre o agente e o servidor, são criptografas com a chave secreta, conforme foi mostrado na Figura 3.1.

A leitura das mensagens durante o processo de localização de chave baseia-se nas chaves públicas trocadas entre o agente e o XKMS Server na etapa de registro, ou seja, todas as mensagens transmitidas durante essa operação são criptografadas com a chave privada dos seus respectivos emissores e só podem ser lidas mediante a chave pública destes. Sendo assim, para efetuar uma comunicação segura, o agente requisita a chave pública do destinatário e aguarda uma resposta do servidor. O XKMS Server, através da identificação do proprietário, retorna a chave pública que é armazenada na base e dados do agente. A Figura 3.4 ilustra o funcionamento dessas operações no XKMS Server.

O XKMS Server é baseado em um modelo de gerenciamento de chaves que cumpre com os requisitos de geração, troca, armazenamento, proteção, utilização e verificação de chaves. Entretanto, esse servidor de chave não incorpora mecanismos que definem o tempo de vida útil das chaves e sua conseqüente substituição por parte dos agentes [OLIVEIRA, 2006]. A renovação de chave é um importante recurso de segurança, pois concede aos usuários os seguintes benefícios:

- Protege o sistema contra envio de mensagens criptografadas com chaves públicas comprometidas;
- Fornece mecanismos para restringir a quantidade de dados expostos, quando uma chave pública é comprometida;
- Fornece caminhos transparentes para mudança de algoritmos de criptografia ou comprimento de chaves.

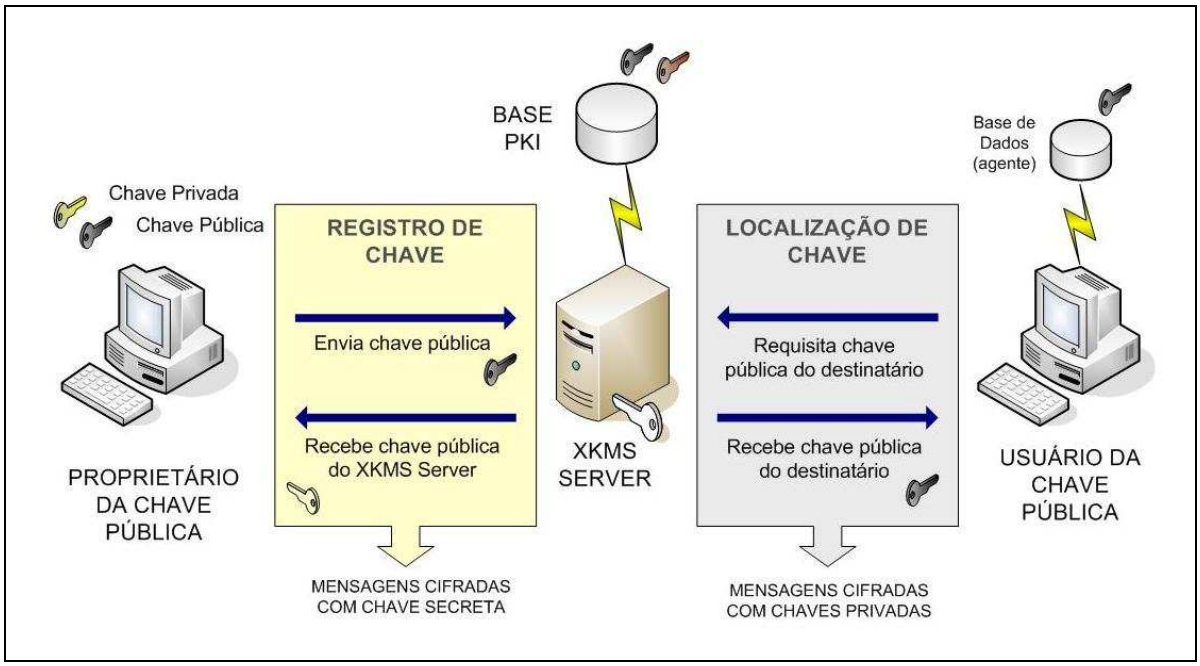

Figura 3.4 – Funcionamento do XKMS Server.

Devido a esse problema, a solução propõe um modelo de segurança onde todas as chaves armazenadas no XKMS Server terão um tempo de vida útil definido pelo próprio servidor, através do uso de declarações digitais denominadas *timestamp*. A adoção de selos de tempo (*timestamp*) às chaves públicas dos agentes garante um reforço à segurança na comunicação dos agentes, pois mesmo que um indivíduo não autorizado obtenha êxito no rastreamento e decifragem de uma chave do sistema, a leitura da mensagem criptografada com esta chave poderá ser automaticamente bloqueada pelo seu proprietário (agente), caso ela esteja fora do prazo de validade. O XKMS Server tem a responsabilidade de manter atualizado o repositório de chaves temporárias do sistema, fornecendo aos usuários apenas chaves públicas que estiverem dentro do prazo de validade.

Timestamp é um conjunto de técnicas que nos habilita a determinar se certo documento digital foi criado ou assinado antes de um dado tempo [WOUTERS, 2002]. Embora a probabilidade de comprometimento das chaves seja pequena, os *timestamps* são úteis por outros motivos: eles garantem a integridade das chaves e protegem contra reutilização de chaves de comunicação anteriormente comprometidas.

Baseado nisso, o uso de *timestamp* anexado à chave pública dos agentes pode exercer um importante papel na clássica infra-estrutura de chave pública PKI. A Figura 3.5 mostra como as operações do XKMS Server são executadas com a inclusão de *timestamp*.

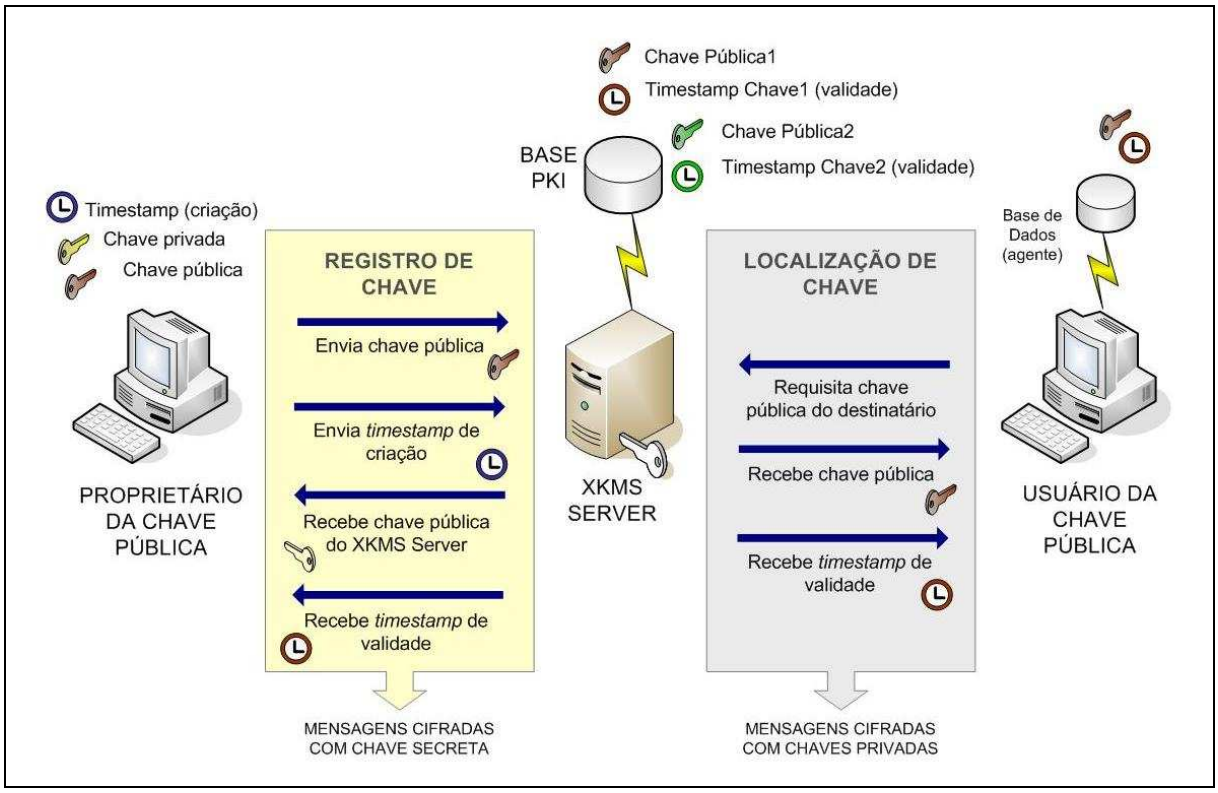

Figura 3.5 – Funcionamento do XKMS Server com a solução de *timestamp*.

Antes de executar o registro de chave, o agente inicia o processo de geração do par de chaves criptográficas e cria um *timestamp* desse processo, que representa a data de criação da chave pública. Em seguida, o agente envia a chave pública e o *timestamp* para o XKMS Server, sendo que esse *timestamp* será usado para calcular o tempo limite em que a chave permanecerá armazenada no repositório do servidor, para ser utilizada pelos outros agentes do sistema, conforme é mostrado na Figura 3.5. Esse tempo calculado corresponde ao *timestamp* de expiramento da chave. Em seguida, o XKMS Server retorna ao agente o *timestamp* de expiramento, que servirá de alerta para execução de um novo registro de chave. Feito isso, o identificador do agente, a *timestamp* de expiramento e a chave pública são finalmente armazenados na base de dados PKI do servidor.

Um agente que quiser enviar uma mensagem segura utilizando a chave pública de outro agente, deverá antes solicitar a chave junto ao XKMS Server. Para isso, uma requisição de chave é enviada para o XKMS Server, e este retorna a chave pública e o seu *timestamp* de expiramento, conforme é mostrado na figura 3.5. Em seguida, essas informações são armazenadas na base de dados do agente para serem facilmente acessadas e reutilizadas na troca de mensagens futuras. Uma nova requisição de chave ao XKMS Server só poderá ser executada caso o agente não possua a chave do destinatário ou se a chave deste estiver fora do

prazo de validade. Portanto, antes do envio de mensagens, o agente examina se a chave consta na sua base de dados e, se estiver armazenada, verifica através do *timestamp* de expiramento se a chave encontra-se dentro do prazo de validade. Se a chave estiver dentro do prazo de validade, a comunicação ocorrerá normalmente. Caso contrário, o agente requisita a nova chave do destinatário para o servidor.

Podem ocorrer casos em que uma chave é verificada pelo emissor dentro do prazo de validade, mas a mensagem cifrada por ela é lida pelo receptor fora do prazo, gerando inconsistência de verificação. Este problema geralmente ocorre quando há possível congestionamento na rede e faz com que o intervalo de tempo entre a verificação do emissor e a leitura da mensagem pelo receptor seja maior que o intervalo de tempo entre essa mesma verificação e a renovação da chave do receptor. Nesse caso, o receptor descarta a mensagem, emite um alerta de falha de leitura (chave pública expirada) ao emissor e solicita reenvio da mensagem com a nova chave pública.

3.4 Persistência de Mensagens

No campo da programação, persistência refere-se especificamente a capacidade de reter estruturas de dados entre execuções de programas, como por exemplo, um programa de edição de imagens salvando uma seleção complexa ou um processador de texto salvando um histórico de um documento para ser desfeito [RASHID, 2003]. Dessa forma, nos canais de mensagens persistentes, todas as mensagens enviadas são armazenadas em memória estável (tipicamente em disco) de modo que podem, posteriormente, serem recuperadas pelos usuários que não estejam diretamente ligados ao canal no momento da publicação.

Mensagens persistentes são mensagens escritas para *logs* e filas de arquivos de dados através do uso de gerenciadores. Esses gerenciadores, que também são conhecidos como Sistemas de Transferência de Mensagens (*Enterprise Messaging Systems* - EMS) ou MOM (*Message Oriented Middleware*), permitem a comunicação de emissores e receptores, de modo que eles não necessitem estar simultaneamente ativos para coordenarem as suas atividades. Se o gerenciador da fila é reiniciado após uma falha, ele recupera estas mensagens a partir dos dados registrados, e as disponibiliza aos seus destinatários. Mensagens não persistentes são descartadas quando o gerenciador de fila pára, por conseqüência de uma paralisação decorrente de um comando do operador ou devido à falha de alguma parte do sistema.

Como dito anteriormente, as MOMs fornecem mecanismos para integração das aplicações de maneira flexível e fracamente acoplada, provendo entrega assíncrona de dados entre aplicações e atuando como um elemento intermediário. Dessa forma, eles asseguram a entrega de mensagens e facilitam programadores de aplicações, poupando-lhes de conhecimentos de detalhes de RPC (*Remote Procedure Calls*), protocolos de redes e de comunicação. Entretanto, estas aplicações necessitam de um conjunto de rotinas e padrões (APIs) que possibilitem utilizar as funcionalidades desses gerenciadores. Baseado nisso, a *Sun Microsystems* desenvolveu a especificação JMS (*Java Message Service*) [JMS, 2008] para fornecer um meio de programas em Java acessarem essas MOMs.

JMS é um conjunto de interfaces e semânticas associadas que define como clientes JMS acessam as facilidades de um sistema de transferência de mensagens. O JMS da *Sun Microsystems* fornece um padrão de portabilidade para programas Java enviarem e receberem mensagens através de produtos MOM. A portabilidade de JMS é assegurada graças a API JMS que é fornecida como um conjunto de interfaces. Produtos que queiram disponibilizar funcionalidades JMS fornecem um provedor que implementa estas interfaces.

Os objetivos do JMS, como declarado na especificação, são [JMS, 2008]:

- Definir um conjunto simples de conceitos de transferência de mensagens e capacidades;
- Minimizar os conceitos que o programador deve aprender para utilizar transferência de mensagens;
- Minimizar a portabilidade de transferência de mensagens;
- Minimizar o trabalho necessário para implementar um provedor;
- Fornecer interfaces clientes para ambos os domínios *point-to-point* e *publish/subscribe*.

A solução de segurança também sugere um segundo modelo de comunicação segura, fazendo uso de MOM para prover mensagens persistentes. O emprego desse tipo de mensagem visa promover proteção e integridade das informações enviadas entre os agentes do IDS, caso haja uma possível falha em algum componente (agente) do sistema. Desse modo, qualquer mensagem enviada terá seu conteúdo preservado e todos os agentes que integram o sistema terão suas mensagens disponíveis tão logo eles se recuperem da falha.

A Figura 3.6 mostra como se dará a comunicação entre um agente emissor e um receptor da mensagem, dentro do sistema.

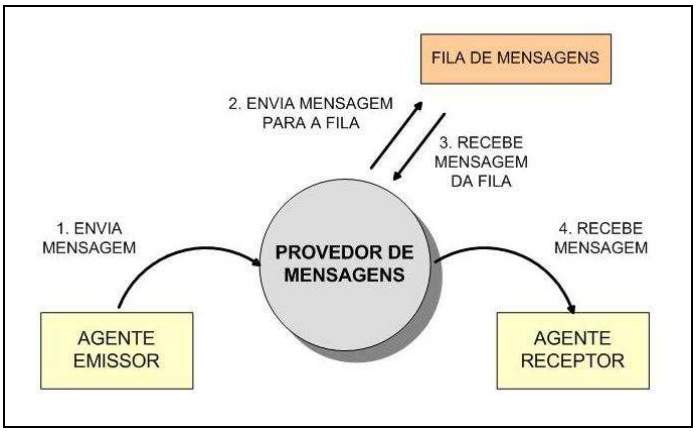

Figura 3.6 – Esquema geral da solução de persistência.

O mecanismo proposto consiste na adição de módulos de tratamento de mensagens integrado aos agentes. Um módulo é responsável pelo envio das mensagens dos agentes para a fila do provedor e o outro módulo é responsável em identificar e receber essas mensagens da fila e disponibilizá-las aos seus destinatários. A comunicação é, portanto, de forma indireta e a mensagem permanece armazenada na fila com um identificador e seu conteúdo original criptografado. A Figura 3.7 ilustra a solução proposta de persistência.

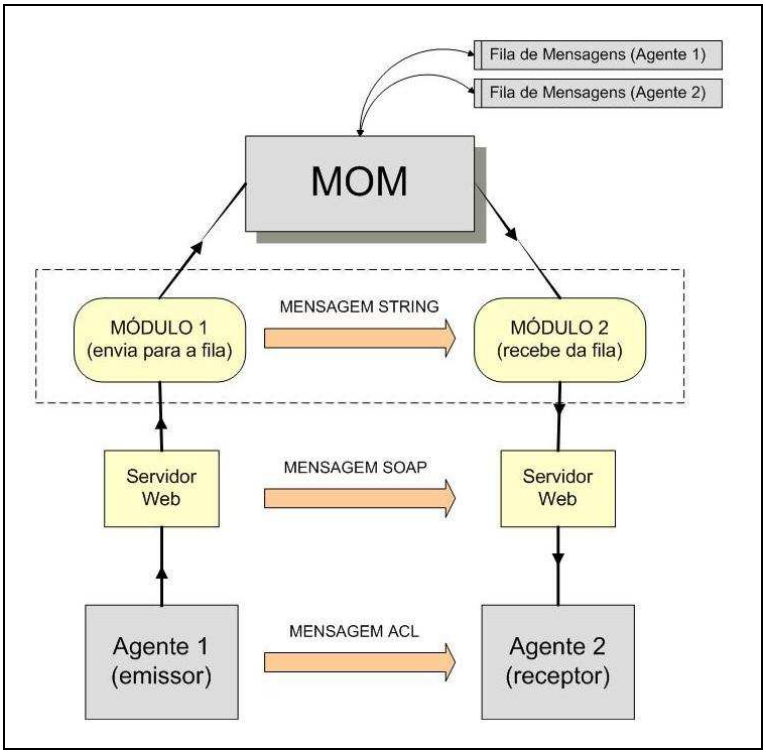

Figura 3.7 – Esquema da solução proposta de persistência.

De acordo com a Figura 3.7, o processo de envio de uma mensagem de um agente A para um agente B, passa por diferentes etapas. Primeiramente, o agente emissor cria uma mensagem no formato de ACL (*Agent Communication Language*), que é o formato padrão lido pelos agentes. Em seguida, essa mensagem é encapsulada numa mensagem SOAP (*Simple Object Access Protocol*), que é o padrão de mensagens utilizados pelos *Web Services*, cuja codificação, utiliza o padrão XML. Finalmente, a mensagem é convertida num formato reconhecido pelo provedor (*String*) e armazenada na fila. O agente receptor captura a mensagem diretamente da fila e as converte de volta ao seu formato original, utilizando os mesmos procedimentos de envio para fila, porém executados de forma inversa.

A especificação XKMS (*XML Key Management Specification*), que define protocolos para operações de registro e distribuição de chaves públicas, tem seu foco nas mensagens SOAP. Como o XKMS Server foi desenvolvido a partir dessa especificação, as mensagens SOAP foram integradas a troca de mensagens para que a especificação forneça mecanismos de segurança em esquemas de comunicação não segura que usem mensagens XML [OLIVEIRA, 2006].

O modelo de persistência proposto assegura a integridade na manipulação das mensagens em um sistema distribuído tal como um IDS executado por agentes. Sendo assim, as mensagens que trafegam por esse sistema terão confiabilidade garantida, pois seu conteúdo jamais se perderá caso haja uma pane do sistema.

3.5 Conclusão

Este capítulo abordou as características gerais de cada mecanismo de segurança desenvolvido para integrar a solução proposta nesta dissertação.

O modelo proposto apresentou quatro mecanismos para prover segurança e confiabilidade à comunicação dos agentes: proteção de informações de configuração, autenticação e autorização, gerenciamento de chaves com *timestamps* e persistência de mensagens.

Para garantir proteção às informações vitais do XKMS Server (IP, MAC e parâmetros da chave secreta), que são compartilhadas com os agentes, a solução apresentou um mecanismo de geração e cifragem dinâmica do arquivo de configuração, sendo executado pelo servidor. Desse modo, o acesso a leitura desse arquivo fica restrito aos agentes que possuírem a chave pública de configuração, que somente é disponibilizada para aqueles pertencentes ao sistema. Para a segurança do XKMS Server, a solução integrou mecanismos

de autenticação e controle de acesso às funções do servidor, limitando o acesso a esses recursos (como registro e localização de chaves) somente para usuários legítimos do sistema. Além disso, a solução adotou algumas medidas de segurança para que essas informações confidenciais pudessem ser transmitidas, baseando-se no protocolo de comunicação SSL. Para prover um gerenciamento de chaves mais eficiente ao XKMS Server, a solução apresentou um mecanismo que define um tempo de vida útil (*timestamp*) às chaves públicas armazenadas, criando novas formas de tratamento quanto à validade dessas chaves. Finalmente, a solução integrou um provedor MOM como elemento intermediário na comunicação dos agentes para adquirir persistência na troca de mensagens. Seu objetivo é fornecer confiabilidade às informações que são transmitidas em ambientes suscetíveis a falhas.

4 Modelagem e Implementação da Solução Proposta

Este capítulo tem por objetivo apresentar em um nível mais detalhado os mecanismos de segurança e confiabilidade que integram a solução proposta nesta dissertação. A solução, que envolve os mecanismos de proteção de informações de configuração, autenticação e autorização, gerenciamento de chaves com *timestamp* e persistência de mensagens, é apresentada neste capítulo, em termos estruturais utilizando a modelagem em UML (*Unified Modeling Language*). Portanto, diagramas de pacote, diagramas de classes e diagramas de seqüência do processo de modelagem de cada mecanismo proposto são apresentados como parte central das atividades que levaram a implementação da solução. Finalmente, para validar o modelo da solução proposta, fragmentos do código do protótipo proposto são apresentados.

4.1 Proteção de Informação de Configuração

Esta seção dedica-se às fases de modelagem e prototipagem da solução que envolve o mecanismo de proteção de informação de configuração, que dá auxílio a visualização da sua estrutura e do seu comportamento.

4.1.1 Modelagem

O mecanismo de proteção de informações de configuração, que fornece segurança no acesso às informações sensíveis do XKMS Server, tem seu agrupamento de classes embutido no pacote *prototype.security*, sendo que este pacote possui dependência com os pacotes *jade* (para implementação dos agentes) e *com.verisign* (para implementações das especificações de segurança XML), conforme está ilustrado pelo diagrama de pacotes da Figura 4.1.

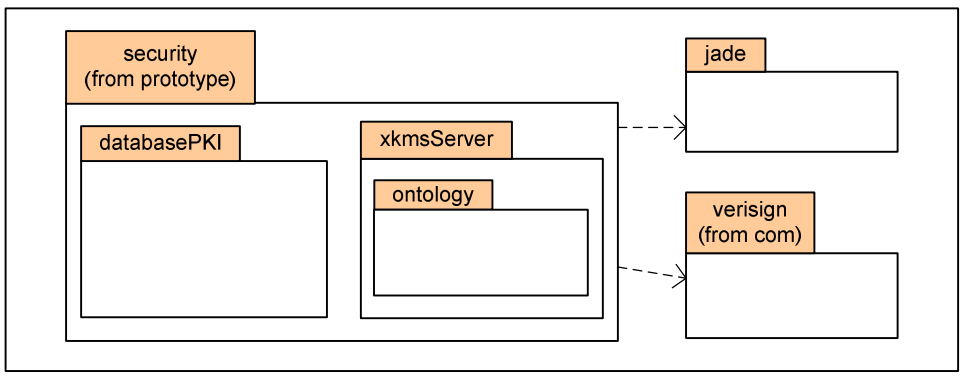

Figura 4.1 – Diagrama de pacotes do mecanismo de proteção de informação.

Na classe *Security* do pacote *prototype.security*, a solução acrescenta importantes modificações na estrutura interna dos métodos utilizados na recuperação das informações de configuração compartilhadas em arquivo, cujo conteúdo envolve o endereço IP e MAC do XKMS Server, e os parâmetros do *Diffie-Hellman* [OLIVEIRA, 2006]. Estas modificações adaptam os métodos da classe *Security* ao mecanismo de segurança proposto. Nesta classe, novos atributos e métodos também são criados, especificamente, para o tratamento das operações de geração, cifragem e armazenamento dessas informações. Sendo assim, essas informações (IP, MAC e valores do *Diffie-Hellman*) consideradas vitais para o sistema, deixam de ser armazenadas em um documento estático e legível, e passam a ser geradas, criptografadas e armazenadas dinamicamente. A Figura 4.2 apresenta o diagrama de classes do mecanismo de proteção de informações de configuração.

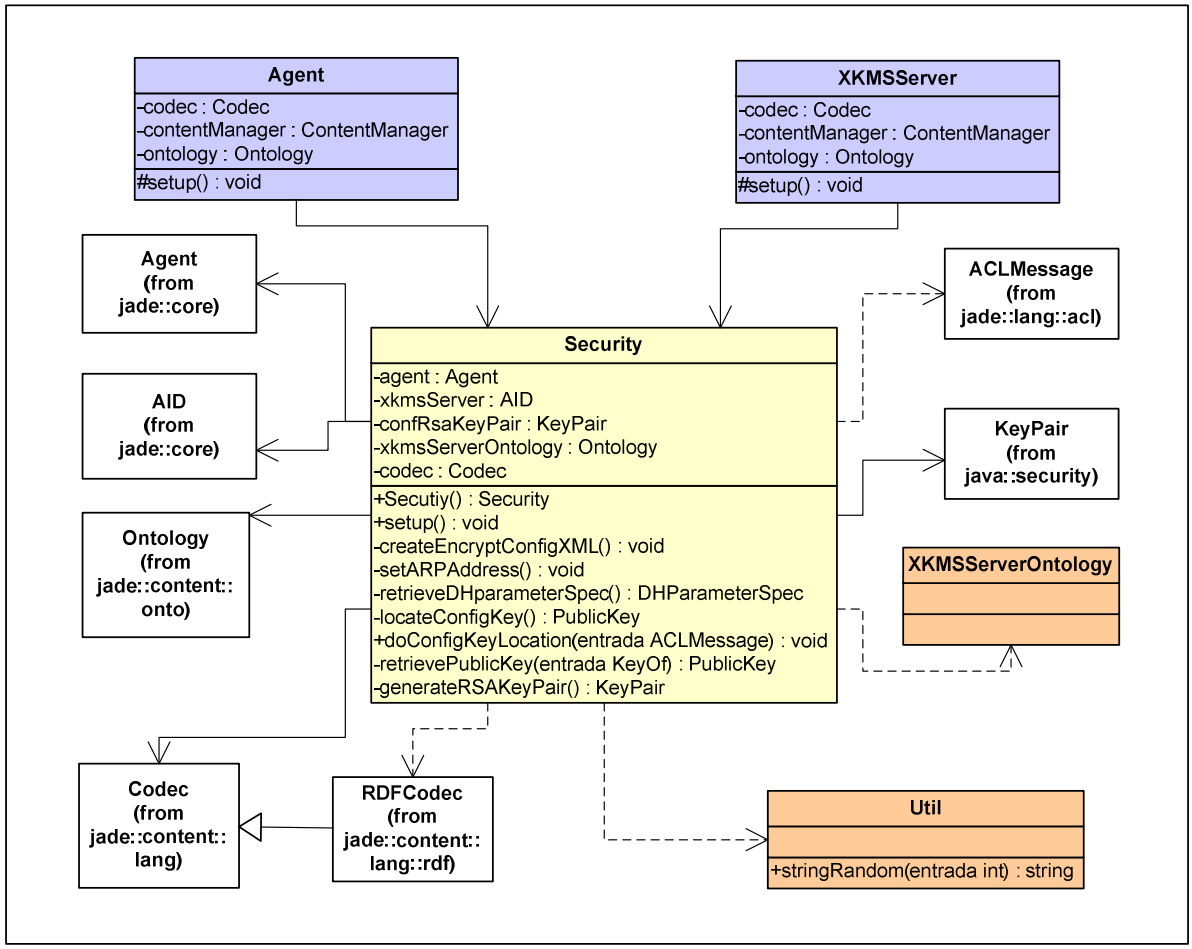

Figura 4.2 – Diagrama de classes do mecanismo de proteção de informação.

A classe *Security* abrange os seguintes atributos e métodos específicos para o mecanismo proposto:

- ***jade.core.Agent agent:*** Atributo que representa a entidade que requisita a chave pública específica para leitura do arquivo de configuração cifrado;
- ***jade.core.AID xkmsServer:*** Atributo que representa a entidade que criptografa as informações de configuração (utilizando a chave privada) e que detém a chave pública para leitura dessas informações;
- ***java.security.KeyPair confRsaKeyPair:*** Atributo que representa o par de chaves público e privado, utilizados na cifragem e decifragem das informações de configuração;
- ***private void createEncryptConfigXML():*** Método responsável pela geração, cifragem e armazenamento dinâmico das informações de configuração por parte do XKMS Server. Essas informações são codificadas e armazenadas num arquivo denominado "conf.xml", localizado no ambiente do servidor;
- ***private void setARPAddress():*** Método responsável pela leitura do IP e MAC do XKMS Server, contidos no arquivo de configuração, e pela definição no sistema operacional, de uma entrada estática correspondente ao MAC do servidor, usado para prevenir o ataque do "homem-do-meio" [OLIVEIRA, 2006];
- ***private DHParameterSpec retrieveDHparameterSpec():*** Método responsável pela leitura dos parâmetros de entrada do algoritmo do *Diffie-Hellman* para geração da chave secreta, que estão armazenados no arquivo de configuração;
- ***private KeyPair generateRSAKeyPair:*** Método responsável pela geração do par de chaves utilizados na proteção do arquivo de configuração;
- ***private PublicKey locateConfigKey():*** Método responsável pela requisição da chave pública de configuração por parte dos agentes. Este método utiliza o canal de comunicação SSL;
- ***public void doConfigKeyLocation (ACLMessage message):*** Método responsável pela localização da chave pública de configuração por parte do XKMS Server. Este método utiliza o canal de comunicação SSL.

Para criptografar as informações de configuração, o XKMS Server aciona o método *setup* da classe *Security*, e este executa internamente os métodos *generateRSAKeyPair* e *createEncryptConfigXML*. O método *createEncryptConfigXML* cria um documento no formato XML, contendo as *tags* de marcação (*markup tags*) e os valores correspondentes às informações compartilhadas. Em seguida, o documento é submetido a um procedimento de

cifragem (utilizando a chave privada gerada no método *generateRSAKeyPair*) para fornecer segurança às informações. Finalmente, o documento é salvo e disponibilizado para os agentes sob a forma de um arquivo XML criptografado. O diagrama de seqüência desse processo é ilustrado na Figura 4.3.

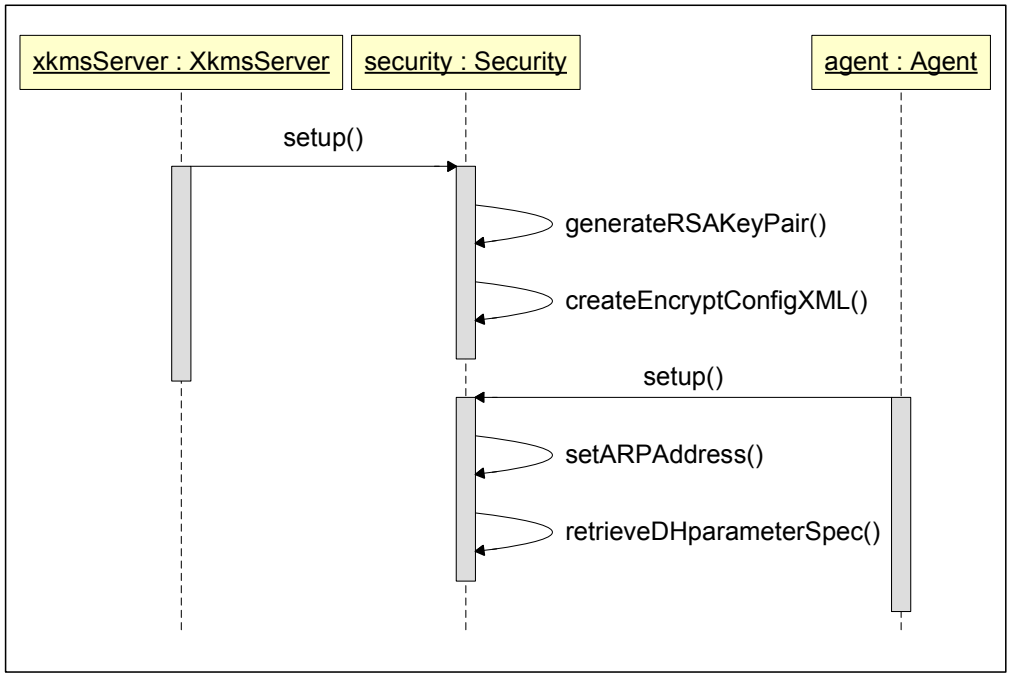

Figura 4.3 – Diagrama de seqüência de criação e recuperação do arquivo de configuração.

A leitura das informações de configuração é executada pelos agentes, através dos métodos *setARPAddress* e *retrieveDHparameterSpec*, conforme é mostrado na Figura 4.3. Cada método executa uma chamada interna ao método *locateConfigKey*, e este disponibiliza a chave pública para decifragem e leitura das informações. O método *setARPAddress* decodifica as informações de endereço do servidor (IP e MAC) e as utiliza para executar o comando ARP (*Address Resolution Protocol*), cuja função é definir uma entrada estática no sistema operacional correspondente ao endereço MAC do XKMS Server. O método *retrieveDHparameterSpec* decodifica e retorna os parâmetros do algoritmo do *Diffie-Hellman* que serão utilizados na geração da chave secreta.

Como elemento extra de proteção, o arquivo de configuração pode ser acessado pelos agentes sobre diferentes formas, seja via canal seguro utilizando protocolo SFTP (*SSH File Transfer Protocol*) ou através de pasta compartilhada com segurança. A escolha do meio mais adequando, fica a cargo do administrador do sistema IDS.

4.1.2 Prototipagem

A Listagem 4.1 apresenta o fragmento de código da classe *Security* contendo os métodos *setup* (linha 3) e *createEncryptConfigXML* (linha 10), conforme é mostrado na Figura 4.2.

Listagem 4.1 – Trecho da classe *Security* de geração do arquivo de configuração.

```
1  public class Security {
2      [...]
3      public void setup() {
4              [...]
5              confRsaKeyPair = generateRSAKeyPair();
6              [...]
7              createEncryptConfigXML();
8              [...]
9      }
10     private void createEncryptConfigXML() {
11             [...]
12             conf = doc.createElement("conf");
13             xkms = doc.createElement("xkms_server");
14             ip = doc.createElement("ip_address");
15             mac = doc.createElement("mac_address");
16             dh = doc.createElement("diffie-hellman");
17             [...]
18             doc.appendChild(conf);
19             conf.appendChild(xkms);
20             xkms.appendChild(ip);
21             ip.appendChild(doc.createTextNode(InetAddress.getLocalHost()
                       .getHostAddress()));
22             xkms.appendChild(mac);
23             mac.appendChild(doc.createTextNode(NetworkInfo.getMacAddress()));
24             conf.appendChild(dh);
25             dh.appendChild(p);
26             p.appendChild(doc.createTextNode("17914[...]"));
27             dh.appendChild(g);
28             g.appendChild(doc.createTextNode("10080[...]"));
29             dh.appendChild(l);
30             l.appendChild(doc.createTextNode("1023"));
31             [...]
32             Encryptor e = new Encryptor(doc, generateDESKey(),
                              AlgorithmType.TRIPLEDES, confRsaKeyPair.getPrivate(),
                              AlgorithmType.RSA1_5);
33             Document encrypt_doc = e.encrypt();
34             DOMSource source = new DOMSource(encrypt_doc);
35             StreamResult result =  new StreamResult("\\\\"
                      + InetAddress.getLocalHost().getHostAddress() +"\\mtp\\conf.xml");
36             transformer.transform(source, result);
37     }
38 [...]
```

As linhas 5 e 7 mostram, respectivamente, a geração das chaves de configuração e a chamada ao método *createEncryptConfigXML* que gera o arquivo de configuração cifrado. Este método cria um documento XML, contendo as *tags* que irão identificar as informações a

serem armazenadas, conforme é mostrado nas linhas 12 a 16. Em seguida, cada informação é anexada às *tags* correspondentes (linhas 18 a 30) e o documento é criptografado com a chave privada de configuração (linha 32 e 33). Finalmente, o documento codificado é salvo e armazenado em um arquivo denominado “conf.xml”, conforme é mostrado nas linhas 34 a 36.

O fragmento de código da classe *Securiy* apresentado na Listagem 4.2 contém os métodos *setARPAddress* (linha 3) e *retrieveDHparameterSpec* (17), que são utilizados na recuperação das informações contidas no arquivo de configuração.

Listagem 4.2 – Trecho da classe *Security* de recuperação do arquivo de configuração.

```
1  public class Security {
2      [...]
3      private void setARPAddress() {
4              [...]
5              publicKeyXkms = locateConfigKey();
6              DOMParser parser = new DOMParser();
7              parser.parse("\\\\"+ InetAddress.getLocalHost().getHostAddress()
                       +"\\mtp\\conf.xml");
8              Document doc = parser.getDocument();
9              [...]
10             Decryptor dec = new Decryptor(doc, publicKeyXkms, xpath);
11             Document decrypt_doc = dec.decrypt();
12             ip = decrypt_doc.getElementsByTagName("ip_address").item(0)
                       .getFirstChild().getNodeValue();
14             mac = decrypt_doc.getElementsByTagName("mac_address").item(0)
                       .getFirstChild().getNodeValue();
15             [...]
16     }
17     private DHParameterSpec retrieveDHparameterSpec() {
18             PublicKey publicKeyXkms;
19             if (agent.getAID().equals(xkmsServer))
20                     publicKeyXkms = confRsaKeyPair.getPublic();
21             else
22                     publicKeyXkms = locateConfigKey();
23             DOMParser parser = new DOMParser();
24             parser.parse("\\\\"+ InetAddress.getLocalHost().getHostAddress()
                       +"\\mtp\\conf.xml");
25             Document doc = parser.getDocument();
26             [...]
27             Decryptor dec = new Decryptor(doc, publicKeyXkms, xpath);
28             Document decrypt_doc = dec.decrypt();
29             String parameterP = decrypt_doc.getElementsByTagName("prime_modulus_P")
                       .item(0).getFirstChild().getNodeValue();
30             String parameterG = decrypt_doc.getElementsByTagName("base_generator_G")
                       .item(0).getFirstChild().getNodeValue();
31             String parameterL = decrypt_doc.getElementsByTagName("bit_size_exponent_L")
                       .item(0).getFirstChild().getNodeValue();
32             return new DHParameterSpec(new BigInteger(parameterP),
                       new BigInteger(parameterG),
                       Integer.parseInt(parameterL));
34     }
35 [...]
36 }
```

As linhas 5, 20 e 22 mostram o recebimento da chave pública de configuração, necessária para decodificação das informações. O arquivo de configuração é capturado no ambiente do servidor (linhas 7, 8, 24 e 25) e seu conteúdo é submetido a um processo de decifragem com a chave pública recebida. As linhas 12 e 14 mostram as informações de endereço IP e MAC do XKMS Server, sendo decodificadas para serem utilizadas na execução do comando ARP, que define as entradas estáticas. As linhas 29 a 31 mostram os parâmetros do algoritmo do *Diffie-Hellman* sendo decodificados para serem utilizados na geração da chave secreta.

4.2 Autenticação e Autorização

Esta seção dedica-se às fases de modelagem e prototipagem da solução que envolve o mecanismo de autenticação e autorização.

4.2.1 Modelagem

O mecanismo de autenticação e autorização, que fornece segurança no acesso aos recursos do XKMS Server, tem seu agrupamento de classes embutido no pacote *prototype.jaas*, sendo que este pacote possui dependência com os pacotes *jade* (para implementação dos agentes), *javax.net.ssl* (para implementação do canal de comunicação segura na rede) e *javax.security.auth* (para implementações dos serviços de autenticação e controle de acesso), conforme está ilustrado pelo diagrama de pacotes da Figura 4.4.

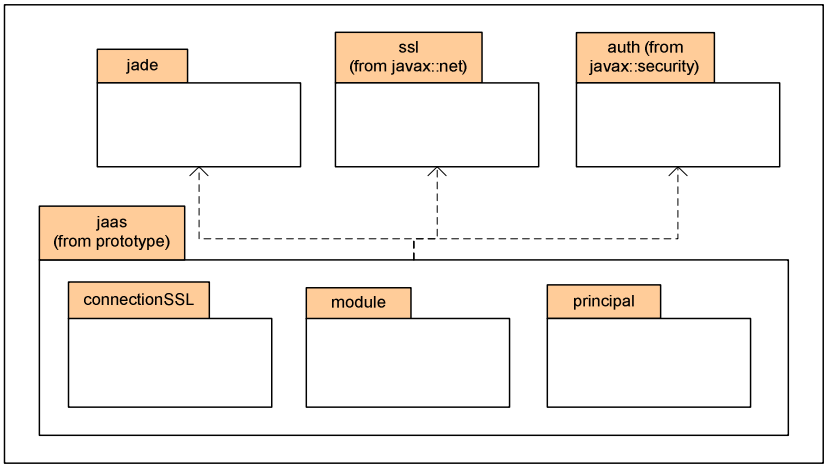

Figura 4.4 – Diagrama de pacotes do mecanismo de autenticação e autorização.

O pacote *prototype.jaas.connectionSSL* contém as classes responsáveis pela criação da conexão segura, processamento das informações confidenciais de autenticação (enviadas pelos agentes para o XKMS Server) e controle das operações executadas pelo servidor após a autenticação (processo de autorização). O pacote *prototype.jaas.module* contém as classes de análise das informações de *login* do agente. O pacote *prototype.jaas.principal* inclui as classes que transformam o agente em um usuário válido para acesso aos recursos (*principal*). As classes definidas para esse mecanismo estão ilustradas pelo diagrama de classes da Figura 4.5.

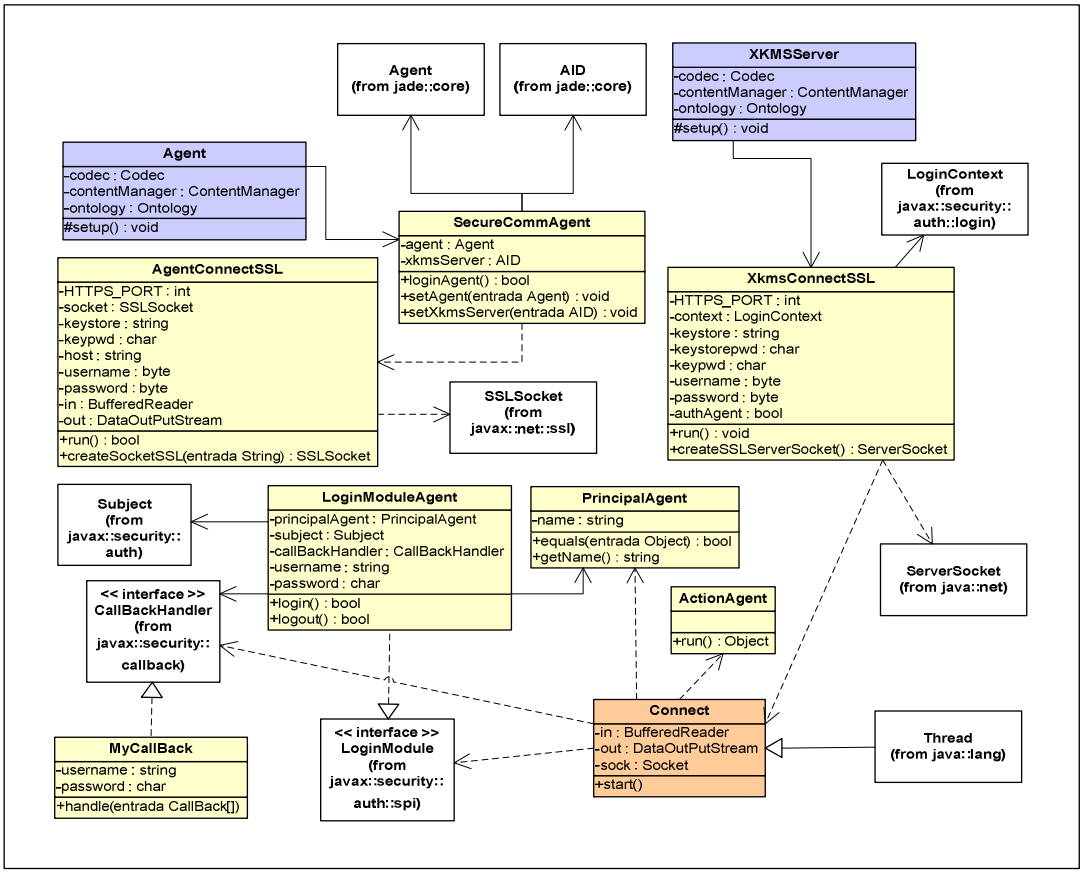

Figura 4.5 – Diagrama de classes do mecanismo de autenticação e autorização.

A classe *SecureCommAgent* do pacote *prototype.jaas* configura as informações de *login* (*username* e *password*) que são transmitidas para o XKMS Server e aciona o serviço de conexão segura. A classe *AgentConnectSSL* do pacote *prototype.jaas.connectionSSL* representa o serviço de conexão segura que é utilizado pelo agente. Essa classe tem como finalidade criar um canal de comunicação que ofereça encriptação, autenticação e integridade às informações de *login* do agente que é transmitidas para o XKMS Server. A classe *XkmsConnectSSL* do pacote *prototype.jaas.connectionSSL* representa o serviço de conexão segura que é utilizado pelo XKMS Server. Essa classe, assim como a classe

AgentConnectSSL, tem como objetivo fornecer um canal de comunicação que ofereça segurança às informações de *login* recebidas dos agentes, além de processar essas informações, para verificar sua autenticidade, e executar as ações para os quais esses agentes foram autorizados. A classe *LoginModuleAgent* do pacote *prototype.jaas.module* representa o mecanismo verificador das informações de *login* que define a autenticidade do agente. A classe *PrincipalAgent* incluída no pacote *prototype.jaas.principal* é utilizada para transformar um agente no usuário válido de acesso (*principal*), ou seja, torná-lo um objeto contendo uma identidade individual que executa uma ação pré-definida. A classe *ActionAgent* do pacote *prototype.jaas* representa as ações privilegiadas executadas pelos agentes e autorizados pelo XKMS Server.

Na classe *XkmsConnectSSL* existem duas classes internas denominada *Connect* e *MyCallBack*, pertencentes ao pacote *prototype.jaas.connectionSSL*. A classe *Connect* é a classe que efetivamente realiza o serviço de comunicação segura. Seu objetivo é criar o *LoginContext* necessário para a autenticação, efetuar a autenticação e executar a ação privilegiada da entidade (*subject*) autenticada, através de um fluxo de execução (*thread*) que se conecta com o cliente. A classe *MyCallBack* é utilizada como meio de transmissão do *username* e do *password* do agente para o mecanismo verificador de *login* do servidor, representado pela classe *LoginModuleAgent*.

De acordo com a Figura 4.5, as classes *XkmsConnectSSL* e *AgentConnectSSL* são definidas como as classes principais do mecanismo. Sendo assim, tem-se uma breve descrição de alguns dos seus atributos e métodos, que são considerados importantes para a solução:

- ***int HTTPS_PORT:*** Atributo que representa a porta de comunicação que será utilizada na transmissão das informações confidenciais;
- ***javax.security.auth.login.LoginContext context:*** Atributo que representa o *LoginContext* necessário para a autenticação;
- ***java.lang.String keystore:*** Atributo que representa o local de armazenamento dos certificados. O *keystore* do XKMS Server deve possuir o certificado de todos os agentes do IDS;
- ***char[] keystorepwd:*** Atributo que representa o *password* de acesso ao *keystore* do XKMS Server;
- ***char[] keypwd:*** Atributo que representa o *password* de acesso a um determinado certificado;

- ***boolean authAgent:*** Atributo que obriga ou não o agente a se autenticar para acessar o canal seguro. Para o mecanismo proposto, seu valor é configurado para "*true*";
- ***java.net.Socket socket:*** Atributo que representa o *socket* seguro por meio do protocolo SSL, criado para o agente;
- ***public ServerSocket createSSLServerSocket():*** Método responsável pela configuração de um *socket* seguro para o agente. Esta configuração requer o certificado do servidor e obriga a autenticação do agente;
- ***private SSLSocket createSSLSocket (String host):*** Método responsável pela configuração de um *socket* seguro para o XKMS Server. Esta configuração requer o certificado do servidor;
- ***public void/boolean run():*** Método responsável pela ação de autenticação e autorização de um modo geral. Este método abstrai a complexidade que envolve um procedimento de autenticação e controle de acesso aos recursos do XKMS Server.

Para o XKMS Server executar uma operação de autenticação do agente, ele primeiro necessita criar uma conexão segura. Para isso, ele aciona o método *run* pertencente à classe *XkmsConnectSSL* e este configura um *socket* seguro para o servidor através do método *createSSLServerSocket*. Em seguida, ele aguarda até que um agente se conecte a ele e forneça suas informações de *login*, através do método *accept()* pertencente a classe *ServerSocket*. Este procedimento está ilustrado pelo diagrama de seqüência da Figura 4.6.

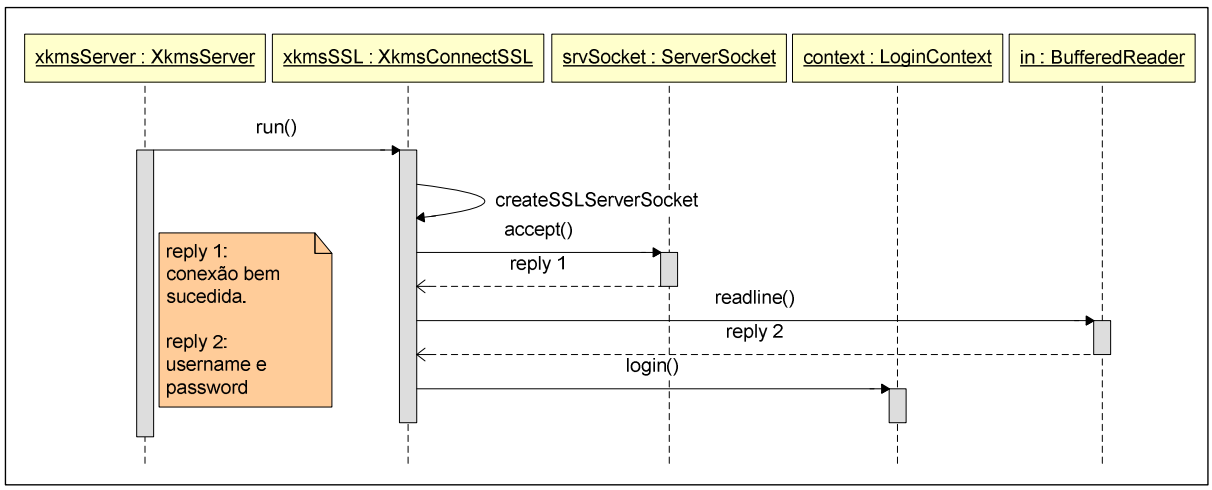

Figura 4.6 – Diagrama de seqüência de verificação de *login* do agente.

Quando o agente se conecta com o XKMS Server, este captura as informações de *login*, através do método *readline()* da classe *BufferedReader*, e o processo de autenticação dessas informações é finalmente executado, através de uma chamada ao método *login* da classe *LoginContext*, conforme também é mostrado na Figura 4.6.

Par um agente executar uma operação de envio de *login* para o XKMS Server, este aciona o método *loginAgent* da classe *SecureCommAgent*, que preenche suas informações confidenciais (*username* e *password*) e as envia para o servidor através do método *run* pertencente a classe *AgentConnectSSL*. Em seguida, uma conexão segura é criada utilizando o método *createSSLSocket* e os dados são finalmente enviados ao servidor, através do método *write* pertencente a classe *DataOutPutStream*. Este processo é mostrado no diagrama de seqüência da Figura 4.7.

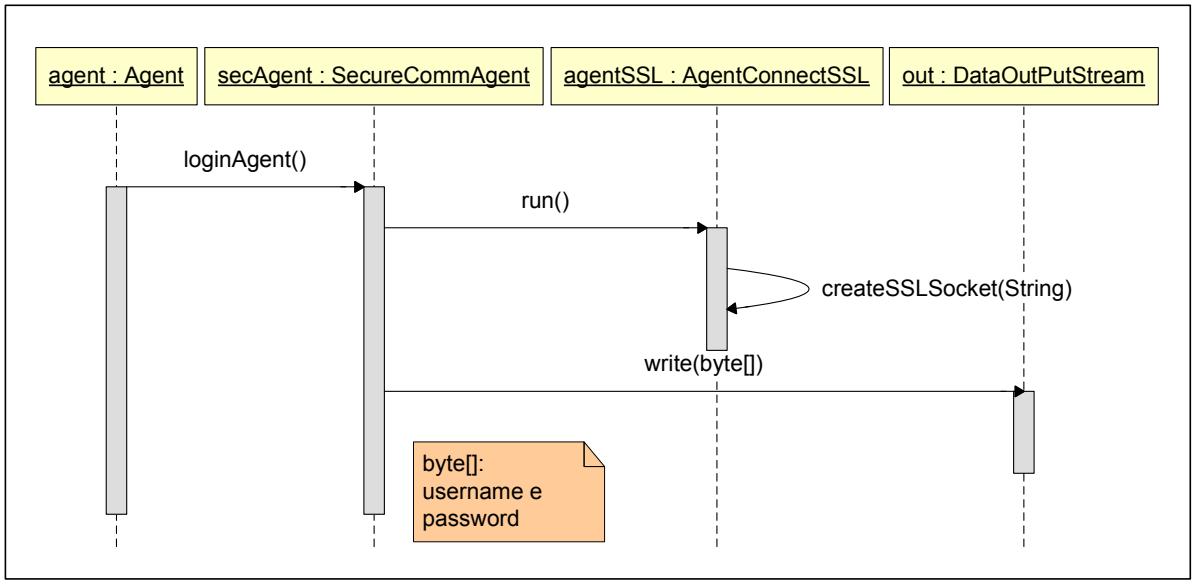

Figura 4.7 – Diagrama de seqüência de envio do *login* do agente.

4.2.2 Prototipagem

O fragmento de código da Listagem 4.3 representa a classe *XkmsConnectSSL* e a classe interna *Connect* (linha 14) contendo o método *run* (linha 3). Este fragmento mostra como o processo de verificação do *login* é implementado no protótipo.

Listagem 4.3 – Trecho da classe *XkmsConnectSSL* de verificação do *login* do agente.

```
1  public class XkmsConnectSSL {
2      [...]
3      public void run() {
4         ServerSocket listen = null;
5         try {
6              ServerSocket listen = createSSLServerSocket();
7              Socket client = listen.accept();
8              Connection connect = new Connection(client);
9              listen.close();
10        }catch(Exception e){
11        [...]
12        }
13     }
14     class Connect extends Thread {
15        [...]
16        try {
17             in = new BufferedReader(new InputStreamReader
                      (socketclient.getInputStream()));
18             out = new DataOutputStream(socketclient.getOutputStream());
19        }catch (IOException e)
20             this.start();
21     }
22       public void run(){
23          try {
24             [...]
25             msg = in.readLine();
26             [...]
27             username = msg.getBytes();
28             password = msg.getBytes();
29             [...]
30             lc = new LoginContext("Sample", new MyCallbackHandler(username, password));
31             [...]
32             try {
33                    lc.login();
34             } catch (LoginException le) {
35                    out.write(error);
36                    [...]
37             }
38             Subject mySubject = lc.getSubject();
39             [...]
40             out.write(sendKeyConfig);
41             PrivilegedAction action = new ActionAgent(out);
42             Subject.doAsPrivileged(mySubject, action, null);
43             in.close();
44             out.close();
45             socketclient.close();
46          }catch (Exception e)
47       }
48     }
49 }
```

A linha 6 mostra a chamada ao método *createSSLServerSocket* que configura o *socket* do servidor, baseando-se nas suas credenciais (certificado do servidor) e que obriga autenticação do agente. A linha 7 mostra o método *accept* que deixa o servidor em estado de espera, aguardando a conexão de um agente. Quando um agente se conecta, seus dados confidenciais são lidos (linha 25) e enviados ao mecanismo de verificação de autenticidade,

conforme é mostrado na linha 30. Em seguida, o processo de verificação é iniciado (linha 33) e, caso a autenticidade do agente seja confirmada, o agente é transformado em um usuário válido de acesso (linha 38). Finalmente, o XKMS Server configura o recurso (tipo de ação do servidor) que será disponibilizado para o agente (linhas 40 e 41) e executa a ação baseando-se no seu *principal*, conforme é mostrado na linha 42.

A Listagem 4.4 apresenta o fragmento de código da classe *AgentConnectSSL* que inclui o método *run* (linha 6) responsável pelo envio do *login* do agente.

Listagem 4.4 – Trecho da classe *AgentConnectSSL* de envio do *login* do agente.

```
1  public class AgentConnectSSL{
2       [...]
3       public AgentConnectSSL(byte[] username, byte[] password,
                        String certpass, String host){
4               [...]
5       }
6       public boolean run(){
7               [...]
8               try{
9                       socket = createSSLSocket(host);
10              }catch(Exception e)
11              try{
12                      DataOutputStream out = new DataOutputStream(socket.
                                getOutputStream());
13                      BufferedReader in = new BufferedReader(new InputStreamReader(socket.
                                getInputStream()));
14                      out.write(username);
15                      out.write(password);
16                      out.flush();
17                      respServer = in.read();
18                      in.close();
19                      out.close();
20                      [...]
21                      socket.close();
22              }catch(Exception e)
23      return success;
24      }
25 }
```

A linha 3 contém o construtor da classe que recebe as informações confidenciais do agente para serem processadas. Na linha 9, tem-se o método *createSSLSocket* que configura o *socket* do agente, baseando-se nas credenciais do servidor (certificado do servidor). Em seguida, as informações são escritas e enviadas para o servidor (linhas 14 a 16) e uma resposta é transmitida de volta (linha 17) definindo o sucesso ou o fracasso da operação.

4.3 Gerenciamento de Chaves e *Timestamps*

Esta seção dedica-se às fases de modelagem e prototipagem da solução que envolve o mecanismo de gerenciamento de chaves com *timestamp*.

4.3.1 Modelagem

O mecanismo de segurança para gerenciamento de chaves com o uso de *timestamp* acrescenta alterações significativas à estrutura interna do protótipo de comunicação segura, provendo ao XKMS Server, suporte ao tratamento do tempo de vida útil das chaves armazenadas no repositório. As alterações envolvem um conjunto de classes, com atributos e funções específicas para o mecanismo, e aperfeiçoamento de algumas classes existentes, adaptando-as ao objetivo proposto. O modelo desenvolvido está ilustrado pelo diagrama de pacotes da Figura 4.8.

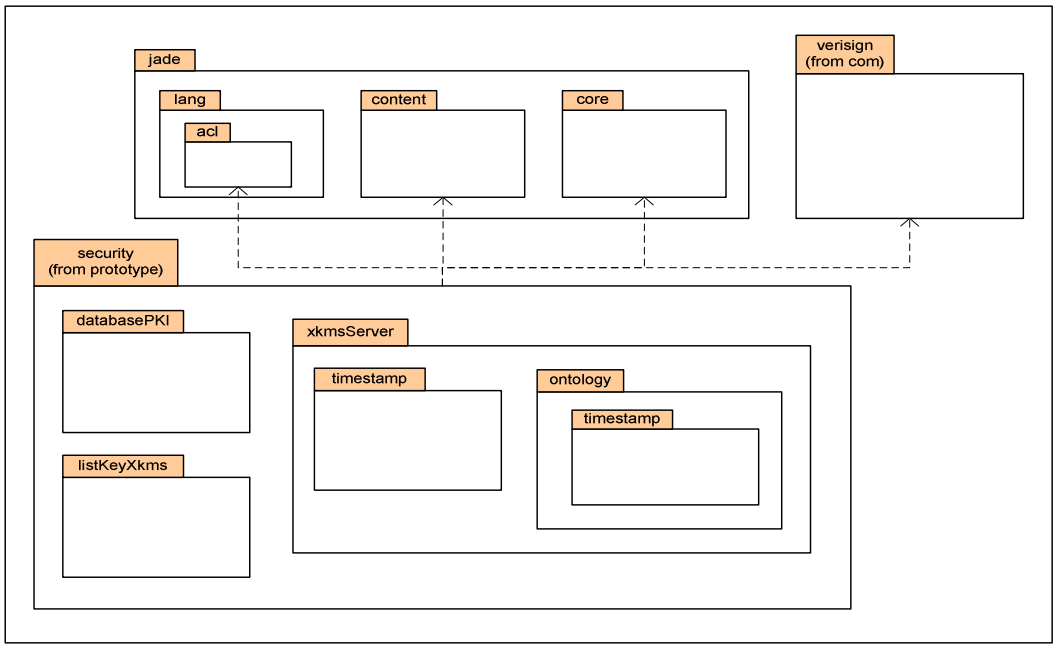

Figura 4.8 – Diagrama de pacotes do gerenciamento de chaves com *timestamp*.

O pacote *prototype.secutity*, que também integra as classes utilizadas nesse mecanismo, é considerado o mais importante do protótipo e possui dependência com os pacotes *jade.lang.acl* (responsável pelo envio de mensagens de registro e localização de chaves), *jade.content* (responsável pela manipulação de ontologias e *codecs* de linguagem de conteúdo), *jade.core* (responsável pela manipulação das representações dos agentes JADE) e *com.verisign*.

O pacote *prototype.secutity.databasePKI* possui as classes que representam uma base de dados PKI. O pacote *prototype.secutity.xkmsServer* inclui as classes que simulam o XKMS Server. Este pacote possui um agente que fará o papel de um servidor de chaves com a responsabilidade de responder às requisições de registro e localização dessas chaves [OLIVEIRA, 2006]. O pacote *prototype.secutity.xkmsServer.ontology* é responsável pela definição da ontologia utilizada para as mensagens ACL, que representam as requisições de registro e localização de chaves. Os pacotes *prototype.secutity.xkmsServer.timestamp* e *prototype.secutity.xkmsServer.ontology.timestamp* contém classes relacionadas ao mecanismo de *timestamp*, que auxiliam, respectivamente, na simulação do XKMS Server e na definição da ontologia.

A Figura 4.9 apresenta o diagrama de classes do mecanismo de gerenciamento de chaves com *timestamp*.

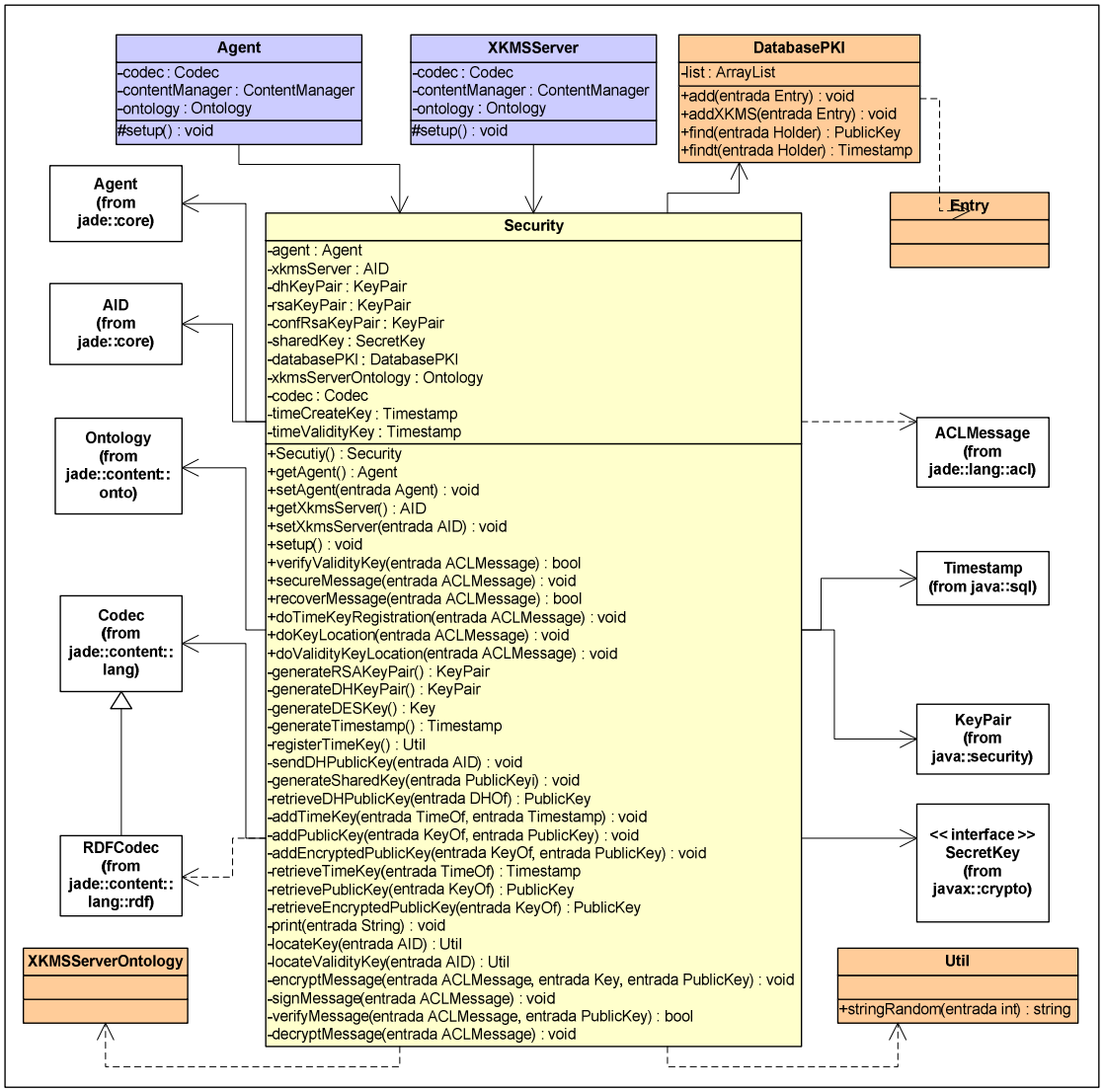

Figura 4.9 – Diagrama de classes do gerenciamento de chaves com *timestamp*.

A classe *Secutity* do pacote *prototype.secutity* é a classe mais importante do mecanismo de gerenciamento de chaves, sendo que ela é utilizada tanto pelos agentes quanto pelo XKMS Server. Esta classe abstrai o esquema inteiro de PKI além de integrar outros mecanismos utilizados na solução. A classe *Security* possui os seguintes atributos:

- ***prototype.security.databasePKI.DatabasePKI databasePKI:*** Representa a base de dados PKI que abriga as chaves públicas e os *timestamps* de expiramento estocados;
- ***jade.core.Agent agent:*** Representa o agente que será o usuário da solução de segurança;
- ***jade.core.AID xkmsServer:*** Representa a identificação do agente que fará a simulação de um XKMS Server. Ele será o receptor de todas as requisições de registro e localização de chaves;
- ***java.security.KeyPair dhKeyPair:*** Representa os parâmetros utilizados no algoritmo do *Diffie-Hellman*;
- ***javax.crypto.SecretKey sharedKey:*** Representa a chave secreta obtida pelo algoritmo do *Diffie-Hellman*;
- ***java.security.KeyPair rsaKeyPair:*** Representa o par de chaves público e privado utilizados na criptografia das mensagens de comunicação;
- ***jade.content.onto.Ontology xkmsServerOntology:*** Representa a ontologia utilizada pelos agentes e pelo XKMS Server nas requisições de registro e localização de chaves publicas e *timestamps*;
- ***jade.content.lang.Codec codec:*** Representa o codificador da linguagem de conteúdo. Como os agentes trocam mensagens através da linguagem XML, o suporte padrão de mensagens utilizado é o RDF (*Resource Description Framework*) [OLIVEIRA, 2006];
- ***java.sql.timestamp timeCreateKey:*** Representa o *timestamp* de criação da chave pública. Este atributo armazena informações relacionadas ao tempo (data, hora, minuto e segundo);
- ***java.sql.timestamp timeValidityKey:*** Representa o *timestamp* de expiramento da chave pública.

As operações de geração, controle de validade, registro e localização de chaves são, em sua maioria, executados dentro da classe *Security*. Sendo assim, ela apresenta os seguintes métodos públicos que são utilizados pelos agentes usuários da solução:

- ***public Security():*** Representa o construtor da classe;
- ***public void setAgent(Agent agent):*** Tem a função de armazenar o agente que executará a comunicação segura;
- ***public void setXkmsServer(AID xkmsServer):*** Responsável em armazenar o AID que identifica o XKMS Server;
- ***public void setup():*** Responsável por gerar o par de chaves público e privado, gerar os valores do algoritmo do *Diffie-Hellman*, gerar o *timestamp* de criação da chave, registrar a chave pública do agente no XKMS Server e receber a chave pública do XKMS Server;
- ***public boolean verifyValidityKey(ACLMessage message):*** Responsável por verificar o *timestamp* de expiramento das chaves públicas estocadas na base de dados do agente. Este método obriga o agente a requisitar uma nova chave pública, caso a chave verificada esteja fora do prazo de validade;
- ***public void secureMessage(ACLMessage message):*** Responsável pela proteção de uma mensagem. Este método envia a mensagem com seu conteúdo original criptografado;
- ***public void recoverMessage(ACLMessage message):*** Responsável em converter uma mensagem de texto cifrado para sua forma de texto legível.

Além disso, a classe *Security* apresenta métodos públicos e privados que são responsáveis pelo tratamento da complexidade interna da troca de mensagens segura entre os agentes e o XKMS Server. Esta classe agrega funções de uso exclusivo do XKMS Server, rotinas restritas aos métodos públicos utilizados pelos agentes, ou comuns a ambos. Dentre as mais importantes, destacam-se:

- ***public void doTimeKeyRegistration(ACLMessage message):*** Responsável pela ação de registro de chave pública e *timestamp* por parte do XKMS Server. Esse método abstrai a complexidade que envolve a segurança e o acesso concorrente às operações de registro;
- ***public void doKeyLocation(ACLMessage message):*** Responsável pela ação de localização de uma chave pública por parte do XKMS Server;

- ***public void doValidityKeyLocation (ACLMessage message):*** Responsável pela ação de localização do *timestamp* de expiramento da chave pública por parte do XKMS Server;
- ***private KeyPair generateRSAKeyPair():*** Responsável pela criação do par de chaves RSA, que representa as chaves pública e privada dos agentes;
- ***private KeyPair generateDHKeyPair():*** Responsável pela criação dos valores públicos e privados do algoritmo do *Diffie-Hellman* que são utilizados na geração da chave secreta;
- ***private Key generateDESKey():*** Auxilia na geração de chaves DES aleatória;
- ***private Timestamp generateTimestamp():*** Responsável pela geração do *timestamp* de criação da chave pública. Esta informação auxilia o XKMS Server no cálculo do tempo de vida útil das chaves públicas armazenadas no repositório;
- ***private Timestamp generateValidityTmp(Timestamp timestamp):*** Responsável pela geração do *timestamp* de expiramento da chave pública por parte do XKMS Server;
- ***private TimeKeyResponse registerTimeKey():*** Responsável por enviar uma requisição de registro de chave pública e seu *timestamp* para o XKMS Server;
- ***private TimeKeyResponse locateKey(AID holder):*** Responsável por enviar uma requisição de localização de uma chave pública para o XKMS Server;
- ***private TimeKeyResponse locateValidityKey(AID holder):*** Responsável por enviar uma requisição de localização do *timestamp* de expiramento para o XKMS Server.
- ***private void encryptMessage(ACLMessage message, Key desKey, PublicKey publicKey):*** Responsável pela cifragem da mensagem transmitida para um agente. Este método utiliza uma chave DES aleatória e a chave pública RSA do agente receptor;
- ***private void signMessage(ACLMessage message):*** Responsável pela assinatura digital da mensagem com a chave privada do agente remetente;
- ***private boolean verifyMessage(ACLMessage message, PublicKey publicKey):*** Responsável por verificar uma mensagem utilizando a chave pública do agente remetente.

O registro da chave pública de um agente usuário da solução se dá através do método *setup* pertencente à classe *Security*. Esse método ativa as funções de geração do par de chaves criptográficas, geração dos valores do algoritmo do *Diffie-Hellman* (utilizados na criação da chave secreta) e geração do *timestamp*, que são representados, respectivamente, pelos métodos *generateRSAKeyPair*, *generateDHKeyPair* e *generateTimestamp*. O *timestamp* gerado no método *generateTimestamp* representa a data de criação da chave pública. Em seguida, o agente troca informações de registro com o XKMS Server através do método *registerTimeKey*. Durante esse processo, a chave pública e o *timestamp* de criação são enviados para o XKMS Server e este retorna a chave pública do servidor e o *timestamp* de expiramento da chave registrada, para serem armazenados na base de dados do agente.

No XKMS Server, a troca de informações de registro é executada pelo método *doTimeKeyRegistration*. Este método se encarrega de receber as informações do agente (chave pública e *timestamp*), efetuar o cálculo validade da chave, através do método *generateValidityTmp* (baseando-se no *timestamp* fornecido pelo agente), enviar a chave pública do servidor, enviar o *timestamp* de expiramento da chave registrada e armazenar a chave pública do agente, com seu *timestamp* de expiramento, no repositório de chaves. A Figura 4.10 representa o diagrama de seqüência de um registro de chave pública.

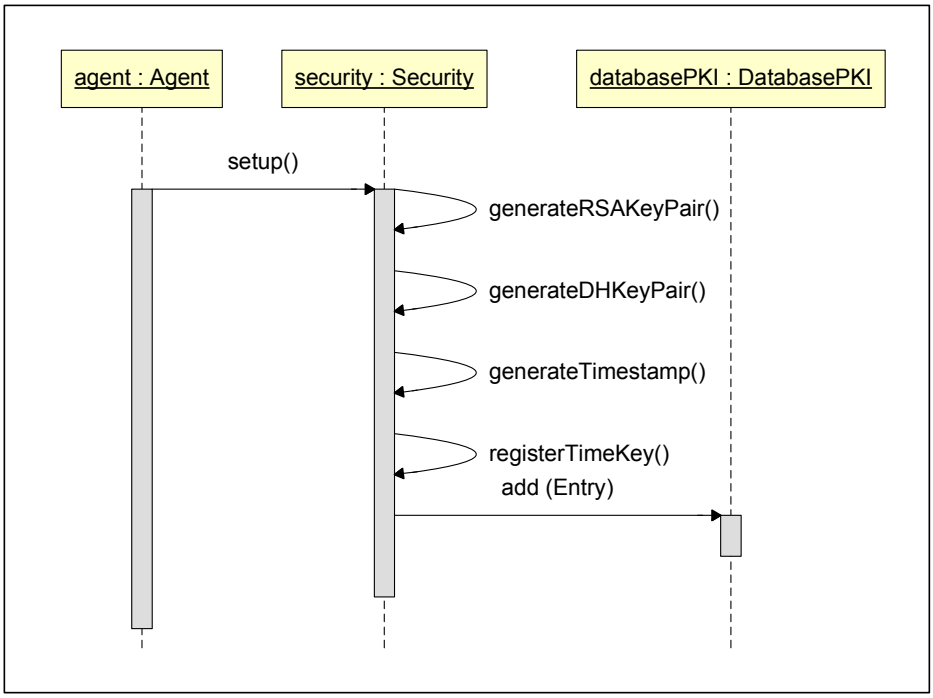

Figura 4.10 – Diagrama de seqüência de registro de chave pública.

O envio de uma mensagem segura, de um agente usuário da solução, é precedido de uma verificação da validade da chave pública do destinatário por parte desse agente, através do método *verifyValidityKey*. Se a chave do destinatário estiver disponível na base de dados PKI do emissor, este método captura o *timestamp* de expiramento dessa chave e compara com o tempo corrente para verificar se corresponde a uma chave pública válida. Caso o agente emissor não tenha a chave do destinatário armazenada, ele executa uma requisição ao XKMS Server, através do método *locateKey* que solicita a chave e o *timestamp* de validade dessa chave. Esse procedimento de solicitação de chave e *timestamp* também ocorrem em casos de comprovação de chaves públicas fora do prazo de validade, onde o agente requisita a nova chave do destinatário ao XKMS Server. A Figura 4.11 representa o diagrama de seqüência de envio de uma mensagem segura.

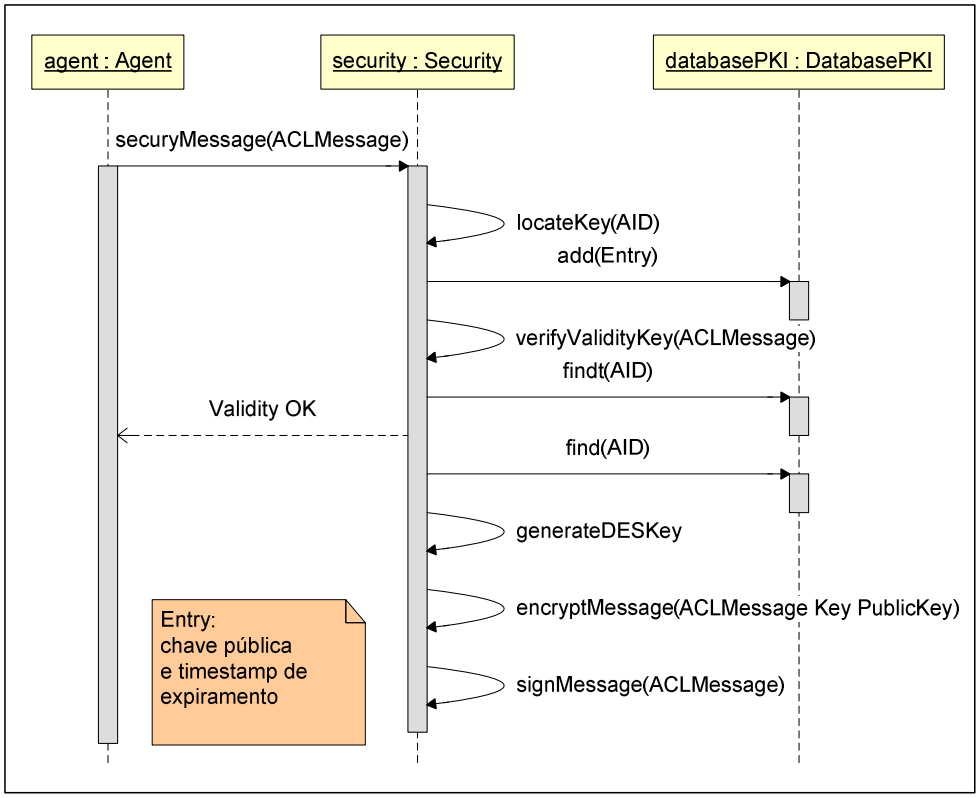

Figura 4.11 – Diagrama de seqüência de envio de mensagem segura.

No XKMS Server, os procedimentos de localização de chaves e localização do *timestamp* de expiramento são executados, respectivamente, pelos métodos *doKeyLocation* e *doValidityKeyLocation*. Essas informações são capturadas no interior do repositório de chaves do servidor, baseando-se nos dados de identificação do proprietário.

4.3.2 Prototipagem

O fragmento de código na Listagem 4.5 apresenta a classe *Security* contendo os métodos *setup* (linha 3) e *registerTimeKey* (linha 14), conforme é mostrado na Figura 4.9.

Listagem 4.5 – Trecho da classe *Security* de registro de chave pública com *timestamp*.

```
1  public class Security {
2      [...]
3      public void setup() {
4              [...]
5              rsaKeyPair = generateRSAKeyPair();
6              dhKeyPair = generateDHKeyPair();
7              timeCreateKey = generateTimestamp();
8              if (!agent.getAID().equals(xkmsServer)) {
9                      [...]
10                     timeKeyResponse = (TimeKeyRegistrationResponse) registerTimeKey();
11                     databasePKI.add(new Entry(timeKeyResponse.getTimestamp(),
                               xkmsServer, timeKeyResponse.getPublicKey()));
12             }
13     }
14     private TimeKeyResponse registerTimeKey() {
15             [...]
16             keyOf.setPublicKey(new prototype.security.xkmsServer.ontology.PublicKey());
17             addEncryptedPublicKey(keyOf, rsaKeyPair.getPublic());
18             message = new ACLMessage(ACLMessage.INFORM);
19             message.setSender(agent.getAID());
20             [...]
21             message.addReceiver(xkmsServer);
22             agent.getContentManager().fillContent(message, keyOf);
23             agent.send(message);

24             message = agent.blockingReceive();
25             keyOf = (KeyOf) agent.getContentManager().extractContent(message);
26             timeKeyResponse = new TimeKeyRegistrationResponse();
27             timeKeyResponse.setPublicKey(retrieveEncryptedPublicKey(keyOf));
28             [...]
29             addEncryptedTimestamp(timeOf, timeCreateKey);
30             message = new ACLMessage(ACLMessage.INFORM);
31             message.setSender(agent.getAID());
32             [...]
33             message.addReceiver(xkmsServer);
34             agent.getContentManager().fillContent(message, timeOf);
35             agent.send(message);

36             message = agent.blockingReceive();
37             timeOf = (TimeOf) agent.getContentManager().extractContent(message);
38             timeKeyResponse.setTimestamp(retrieveEncryptedTimestamp(timeOf));
39             return timeKeyResponse;
40     }
41     [...]
42 }
```

As linhas 5 a 7 mostram, respectivamente, o método gerador de chaves público e privado, gerador dos parâmetros *Diffie-Hellman* e gerador de *timestamp*. A chamada ao método *registerTimeKey* é acionada pelo método *setup* identificado na linha 10. As linhas 16

a 23 mostram os passos de envio da chave pública para o XKMS Server. Na linha 17, a chave pública é criptografada antes de ser enviada para o servidor. As linhas 29 a 35 mostram os passos de envio do *timestamp* de criação da chave para o XKMS Server. Finalmente, as informações da chave pública do XKMS Server, contendo seu valor (linhas 24 a 27) e seu *timestamp* (linhas 36 a 38), são recuperadas e armazenadas no banco de dados do agente, conforme é mostrado na linha 11.

A Listagem 4.6 apresenta o trecho de código contendo o métodos *verifyValidityKey* (linha 3) pertencente a classe *Security*. A identificação do destinatário é recebida na linha 5 e será utilizada na captura do *timestamp* de expiramento. Se a chave pública do destinatário estiver presente na base de dados do emissor, a linha 6 mostra a captura do *timestamp* de expiramento dessa chave. Em seguida, o tempo corrente é capturado (linha 8) e comparado com a validade da chave (linha 9), determinando se corresponde ou não a uma chave pública válida.

Listagem 4.6 – Trecho da classe *Security* de verificação de *timestamp* de expiramento.

```
1  public class Security {
2      [...]
3      public boolean verifyValidityKey(ACLMessage message) {
4              [...]
5              receiver = (AID) message.getAllReceiver().next();
6              receiverValidityKey = databasePKI.findt(receiver);
7              [...]
8              currentTime = generateTimestamp();
9              if(receiverValidityKey.before(currentTime))
10                     return false;
11             return true;
12     }
```

A Listagem 4.7 apresenta o trecho de código contendo os métodos *secureMessage* (linha 3) e *locateKey* (linha 16), pertencentes a classe *Security*. Se a chave pública do destinatário estiver presente na base de dados do emissor, a linha 6 mostra a captura dessa chave. Caso não esteja presente, a chamada ao método *locateKey* é acionada através do método *secureMessage*, conforme é mostrado na linha 9. O método *locateKey* requisita a chave pública ao servidor (linhas 18 a 24) e o *timestamp* de expiramento da chave (linhas 29 a 35). A chave pública (linhas 25 a 28) e o *timestamp* de expiramento (linhas 36 a 39) são recebidos e armazenados na base de dados do agente, conforme mostrado na linha 11. Em posse da chave pública do destinatário, a mensagem é criptografada (linha 13) e assinada (linha 14) para finalmente ser enviada.

Listagem 4.7 – Trecho da classe *Security* de envio de mensagem segura.

```
1  public class Security {
2      [...]
3      public void secureMessage(ACLMessage message) {
4              [...]
5              receiver = (AID) message.getAllReceiver().next();
6              receiverPublicKey = databasePKI.find(receiver);
7              [...]
8              if(receiverPublicKey == null) {
9                      timeKeyResponse = (TimeKeyLocateResponse) locateKey(receiver);
10                     receiverPublicKey = timeKeyResponse.getPublicKey();
11                     databasePKI.add(new Entry(receiverValidityKey, receiver,
                               receiverPublicKey));
12             }
13             encryptMessage(message, generateDESKey(), receiverPublicKey);
14             signMessage(message);
15     }
16     private TimeKeyResponse locateKey(AID holder) {
17             [...]
18             request = new ACLMessage(ACLMessage.REQUEST);
19             request.setSender(agent.getAID());
20             request.addReceiver(xkmsServer);
21             [...]
22             agent.getContentManager().fillContent(request, keyOf);
23             secureMessage(request);
24             agent.send(request);

25             response = agent.blockingReceive();
26             recoverMessage(response);
27             keyOf = (KeyOf) agent.getContentManager().extractContent(response);
28             timeKeyResponse.setPublicKey(retrievePublicKey(keyOf));
29             request = new ACLMessage(ACLMessage.REQUEST);
30             request.setSender(agent.getAID());
31             request.addReceiver(xkmsServer);
32             [...]
33             agent.getContentManager().fillContent(request, timeOf);
34             secureMessage(request);
35             agent.send(request);

36             response = agent.blockingReceive();
37             recoverMessage(response);
38             timeOf = (TimeOf) agent.getContentManager().extractContent(response);
39             timeKeyResponse.setTimestamp(retrieveTimestamp(timeOf));
40             return timeKeyResponse;
41     }
42     [...]
43 }
```

4.4 Persistência de Mensagens

Esta seção dedica-se às fases de modelagem e prototipagem da solução que envolve o mecanismo de persistência de mensagens.

4.4.1 Modelagem

O mecanismo de persistência de mensagens, que integra a solução proposta, fornece confiabilidade à comunicação dos agentes através de uma interface para um software de MOM. Sendo assim, o modelo proposto possui dependência com os pacotes *javax.jms* e *javax.naming*, que funcionam basicamente como adaptadores do *middleware*. O modelo desenvolvido está ilustrado no diagrama de pacotes da Figura 4.12.

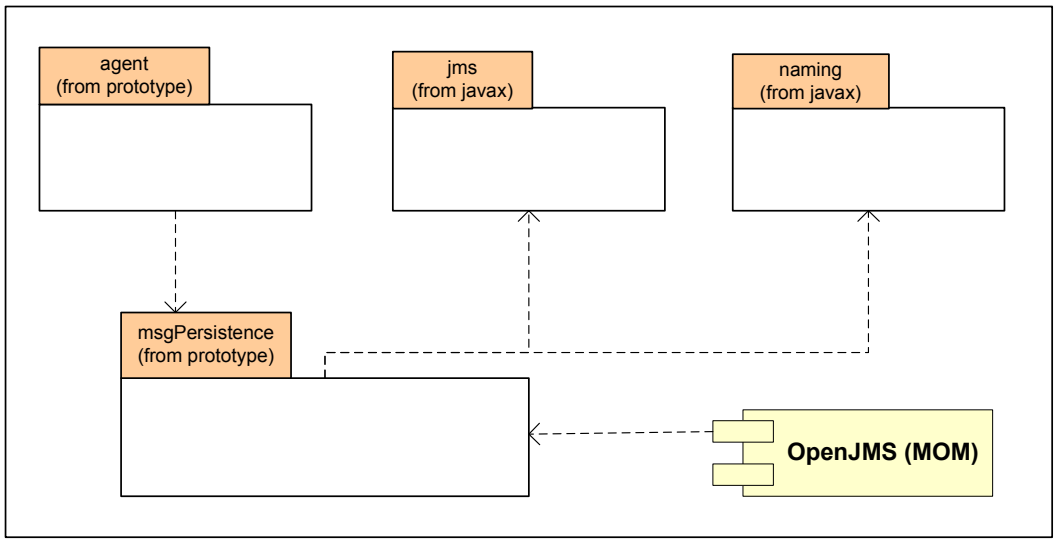

Figura 4.12 – Diagrama de pacotes do mecanismo de persistência.

O pacote *javax.jms* contém um conjunto de classes responsáveis pelo tratamento das mensagens remetidas ou provenientes das filas de mensagens do MOM. Este pacote fornece meios para criar, enviar, receber e ler mensagens de um sistema de transferência de mensagens. O pacote *javax.naming* fornece as classes para acesso aos serviços de nomes (*naming services*). Este pacote cria um contexto JNDI (*Java Naming and Directory Interface*) e configura *factories* de conexões JMS (*connection factory*), tópicos (*topics*) e filas (*queues*) disponíveis para os clientes.

O pacote *prototype.msgPersistence* é formado pelas classes *QueueSend* e *QueueReceive*. A classe *QueueSend* tem a função de estabelecer a conexão com o provedor e enviar as mensagens dos agentes às suas respectivas filas de armazenamento, convertidas no formato aceito por elas. A classe *QueueReceive* tem a função de conectar-se ao provedor, receber as mensagens da fila de armazenamento, convertê-las no formato padrão ACL e retorná-las aos seus destinatários. A Figura 4.13 apresenta o diagrama de classes do mecanismo de persistência.

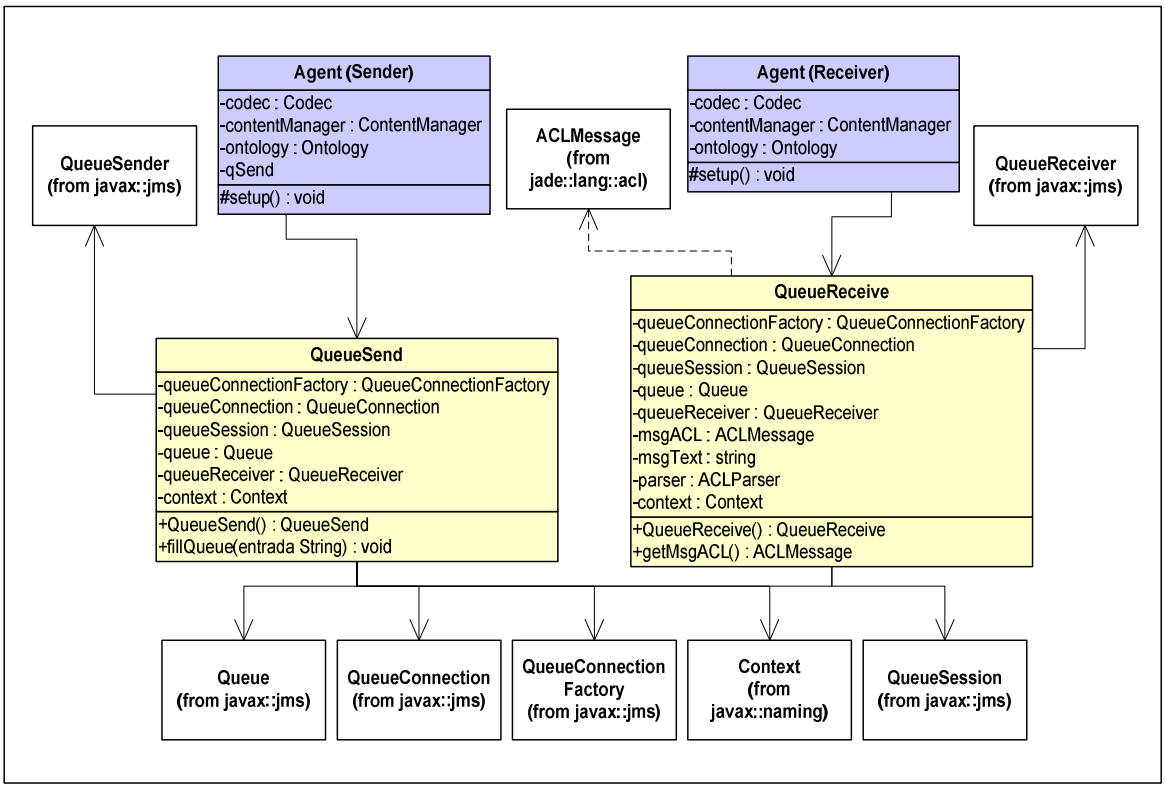

Figura 4.13 – Diagrama de classes do mecanismo de persistência.

Dentre os atributos comuns às duas classes do módulo de persistência se destacam:

- ***javax.jms.QueueConnectionFactory queueConnectionFactory:*** Representa as *factories* de conexões (*queueConnection*) habilitadas para os clientes JMS;
- ***javax.jms.QueueConnection queueConnection:*** Representa uma conexão com a fila do provedor de mensagens;
- ***javax.jms.QueueSession queueSession:*** Representa as configurações de transações e de auto-reconhecimento do recipiente de mensagens do provedor;
- ***javax.jms.Queue queue:*** Representa uma fila do provedor de mensagens habilitada.

Como atributos e métodos particulares da classe *QueueSend* pode-se destacar:

- ***javax.jms.QueueSender queueSender:*** Atributo que representa a fila do emissor que está associada com a fila do provedor de mensagens habilitada;
- ***public void fillQueue(String message):*** Método responsável em preencher a fila do provedor com as mensagens do agente.

Como atributos e métodos particulares da classe *QueueReceive* pode-se destacar:

- ***jade.lang.acl.ACLMessage msgACL:*** Atributo que representa a mensagem convertida no formato *ACLMessage* que é lido pelos agentes;
- ***jade.lang.acl.ACLParser parser:*** Atributo que representa o conversor de mensagens do formato de *String* para *ACLMessage*;
- ***java.lang.String msgText:*** Atributo que representa a mensagem recebida da fila no formato de *String*;
- ***public ACLMessage getMsgACL():*** Método responsável em capturar as mensagens armazenadas na fila do provedor.

Na etapa de envio, o agente emissor (*sender*) remete sua mensagem cifrada ao destinatário (*receiver*) de maneira indireta, de modo que ela é transferida para uma fila de mensagens, através do método *fillQueue* pertencente a classe *QueueSend*. Nesta classe, as configurações iniciais de acesso ao provedor (*JNDI context, connectionFactory, connection, session* e *queue*) são executadas e a mensagem é convertida no formato de *String* para ser armazenado na fila. Em seguida, a mensagem é submetida às configurações de acesso através do método *setText* e enviada para a fila utilizando o método *send* pertencente à classe *QueueSender*. A Figura 4.14 apresenta o diagrama de seqüência desse processo.

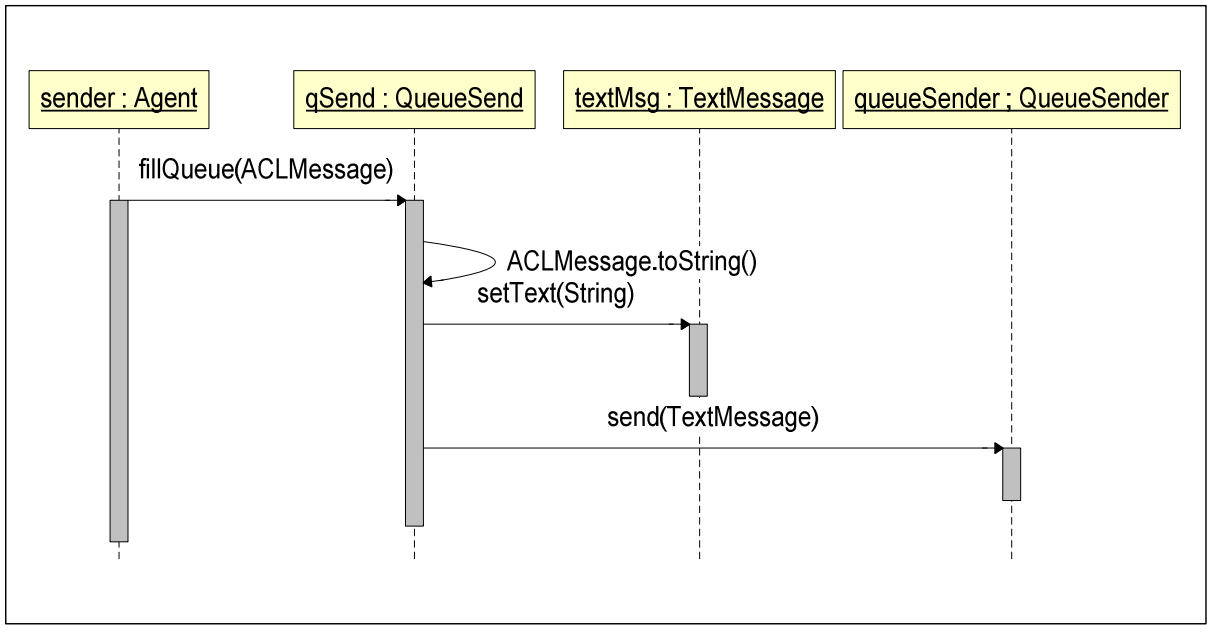

Figura 4.14 – Diagrama de seqüência de envio de uma mensagem persistente.

Na etapa de recebimento, o agente receptor (*receiver*) captura a mensagem armazenada na fila, através do método *getMsgACL* pertencente a classe *QueueReceive*. Nessa classe, as configurações de acesso ao provedor também são executadas e o método *receive* pertencente à classe QueueReceiver é acionado para receber a mensagem da fila, de acordo com essas configurações. Finalmente, a mensagem é convertida de volta ao formato de origem (*ACLMessage*) e disponibilizada para o agente. A Figura 4.15 apresenta o diagrama de seqüência desse processo.

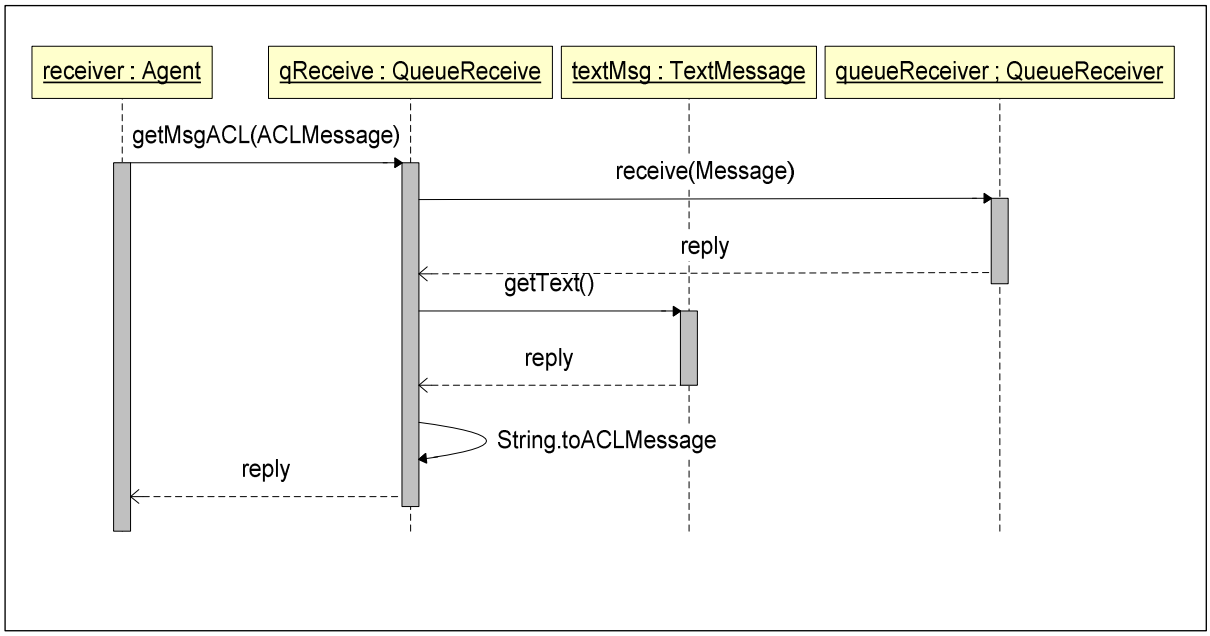

Figura 4.15 – Diagrama de seqüência de recebimento de uma mensagem persistente.

4.4.2 Prototipagem

O fragmento de código na Listagem 4.8 apresenta a classe *QueueSend* contendo o método *fillQueue* (linha 3) responsável pelo envio de mensagens à fila, conforme é mostrado na Figura 4.13.

As linhas 6 a 8 mostram a criação do contexto JNDI que é utilizado nas configurações de acesso às filas do MOM. As linhas 9 a 12 mostram, respectivamente, a recuperação da *factory* de conexão, criação da conexão, criação da seção e recuperação de uma das filas pertencentes ao provedor, que armazenará as mensagens. Em seguida, uma nova fila é criada para o emissor, e esta é associada com a fila do provedor recuperada, conforme é mostrado na linha 13. Finalmente, as linhas 14 a 16 mostram a mensagem sendo enviada para fila.

Listagem 4.8 – Trecho da classe *QueueSend* de envio de mensagem persistente.

```
1  public class QueueSend {
2      [...]
3      public void fillQueue(String msg){
4         try {
5              [...]
6              properties.put(Context.INITIAL_CONTEXT_FACTORY,
                      "org.exolab.jms.jndi.InitialContextFactory");
7              properties.put(Context.PROVIDER_URL,"rmi://192.168.0.176:1099/");
8              Context context = new InitialContext(properties);
9              queueConnectionFactory = (QueueConnectionFactory)context.lookup
                      ("JmsQueueConnectionFactory");
10             queueConnection = queueConnectionFactory.createQueueConnection();
11             queueSession = queueConnection.createQueueSession
                      (false,Session.AUTO_ACKNOWLEDGE);
12             queue = (Queue)context.lookup("queue1");
13             queueSender = queueSession.createSender(queue);
14             TextMessage message = queueSession.createTextMessage();
15             message.setText(msg);
16             queueSender.send(message);
17             [...]
18        }
19        catch (NamingException e)
20             e.printStackTrace();
21        catch (JMSException e)
22             e.printStackTrace();
23     }
24 }
```

A Listagem 4.9 apresenta o fragmento de código, contendo a classe *QueueReceive* e o método *getMsgACL* (linha 24) responsável pela captura das mensagens da fila, conforme é mostrado na Figura 4.6.

As linhas 5 a 7 mostram a criação do contexto JNDI que é utilizado nas configurações de acesso às filas do MOM. As linhas 8 a 11 mostram, respectivamente, a recuperação da *factory* de conexão, criação da conexão, criação da seção e recuperação de uma das filas pertencentes ao provedor, que será utilizada no recebimento das mensagens. Em seguida, uma nova fila é criada para o receptor, e esta é associada com a fila do provedor recuperada, conforme é mostrado na linha 13. Finalmente, a entrega de mensagens é iniciada (linha 14) e cada mensagem recebida (linha 16) é convertida no formato de ACL (linha 20) para ser lida pelo agente.

Listagem 4.9 – Trecho da classe *QueueReceive* de recebimento de mensagem persistente.

```
1  public class QueueReceive {
2       [...]
3       public QueueReceive() {
4               [...]
5               properties.put(Context.INITIAL_CONTEXT_FACTORY,
                        "org.exolab.jms.jndi.InitialContextFactory");
6               properties.put(Context.PROVIDER_URL,"rmi://localhost:1099/");
7               Context context = new InitialContext(properties);
8               queueConnectionFactory = (QueueConnectionFactory)
                        context.lookup("JmsQueueConnectionFactory");
9               queueConnection = queueConnectionFactory.createQueueConnection();
10              queueSession = queueConnection.createQueueSession
                        (false,Session.AUTO_ACKNOWLEDGE);
11              queue = (Queue)context.lookup("queue1");
12              [...]
13              queueReceiver = queueSession.createReceiver(queue);
14              queueConnection.start();
15              [...]
16                      Message msg = queueReceiver.receive();
17                      [...]
18                              messageText = ((TextMessage)msg).getText();
19                      [...]
20                               msgACL = parser.parse(new StringReader(messageText));
21          [...]
22          }
23      }
24      public ACLMessage getMsgACL() {
25              return msgACL;
26      }
27 }
```

4.5 Conclusão

Este capítulo apresentou uma visão detalhada, em termos de modelagem e prototipagem, dos mecanismos de segurança que integram a solução proposta nesta dissertação. Na etapa de modelagem, descreveram-se as funcionalidades de cada mecanismo através de diagramas UML (*Unified Modeling Language*), permitindo analisar sua arquitetura sobre diferentes perspectivas, demonstrando seus aspectos estáticos e dinâmicos. Na etapa de prototipagem, fragmentos de código relacionados às funcionalidades de cada mecanismo foram apresentados e descritos, permitindo identificar no protótipo desenvolvido, certos aspectos que possam garantir a validação da solução proposta.

5 Adaptando e Estendendo o IDS-NIDIA

Este capítulo tem como propósito apresentar a adaptação e extensão do IDS-NIDIA a solução proposta de proteção de informação de configuração, autenticação e autorização, gerenciamento de chaves com *timestamp* e persistência de mensagens, integrando os mecanismos de segurança desenvolvidos à execução e a comunicação dos agentes. Neste capítulo, algumas ilustrações de testes da solução proposta, que comprovam a sua aplicabilidade no IDS-NIDIA são apresentados.

5.1 IDS-NIDIA como Estudo de Caso

A solução proposta teve sua aplicabilidade demonstrada no projeto IDS-NIDIA, sendo que ela foi utilizada na execução e na comunicação dos agentes sensores da camada de monitoramento. Conforme dito anteriormente, agentes sensores desempenham dois papéis no sistema: capturar os pacotes que trafegam na rede (agentes sensores de rede) e coletar *logs* de informações de *host* (agentes sensores de *host*). Em posse dessas informações, os agentes sensores as disponibilizam para os agentes pertencentes à camada de análise, denominados agentes de avaliação do sistema (SEA). Para exemplificar a funcionalidade presente na camada de monitoramento, o mecanismo de coleta de *login* de informações de uma máquina servidora está ilustrado na Figura 5.1.

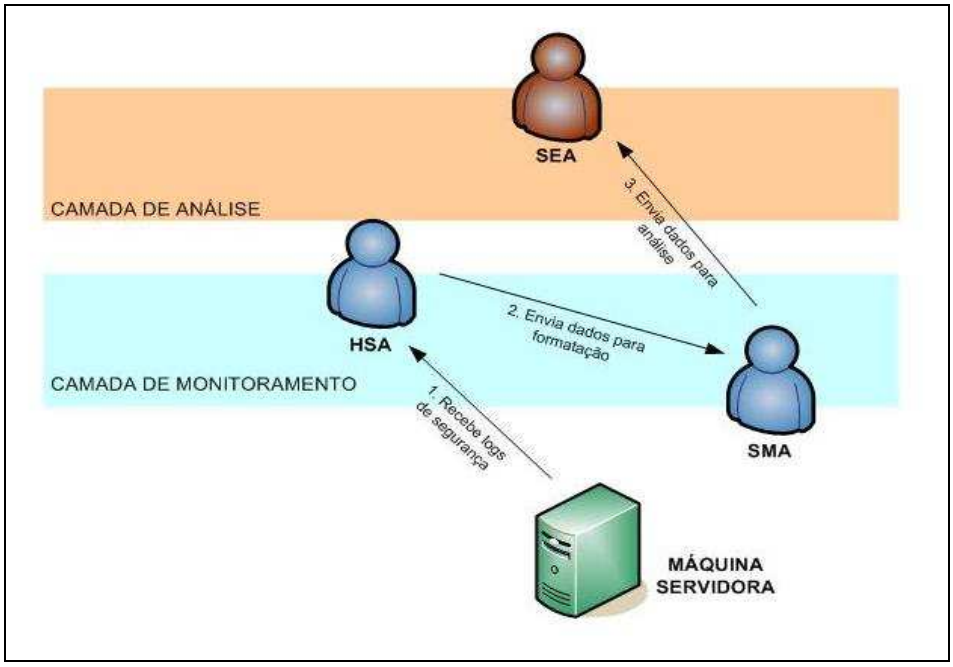

Figura 5.1 – Processo de coleta de dados do IDS-NIDIA.

A Figura 5.1 descreve o agente sensor de *host* (HSA) coletando informações de *log* de uma máquina servidora e enviando essas informações para serem formatadas pelo agente de monitoramento do sistema (SMA). Após a formatação, os dados são enviados para a camada de análise, onde são colhidos e processados pelo agente de avaliação do sistema (SEA).

A solução proposta modifica o processo de coleta de dados da Figura 5.1, provendo segurança e confiabilidade no tráfego das informações. Para adquirir segurança, o modelo cria mecanismos criptográficos para que essas informações de *logs* não sejam lidas, alteradas ou enviadas por indivíduos não autorizados. Para adquirir confiabilidade, o modelo introduz mecanismos de persistência (através de MOM – *Message Oriented Middleware*) para que essas informações não se percam, caso ocorra uma falha do IDS-NIDIA ou da rede. A Figura 5.2 ilustra o esquema geral da solução integrada ao processo de coleta de dados do IDS-NIDIA.

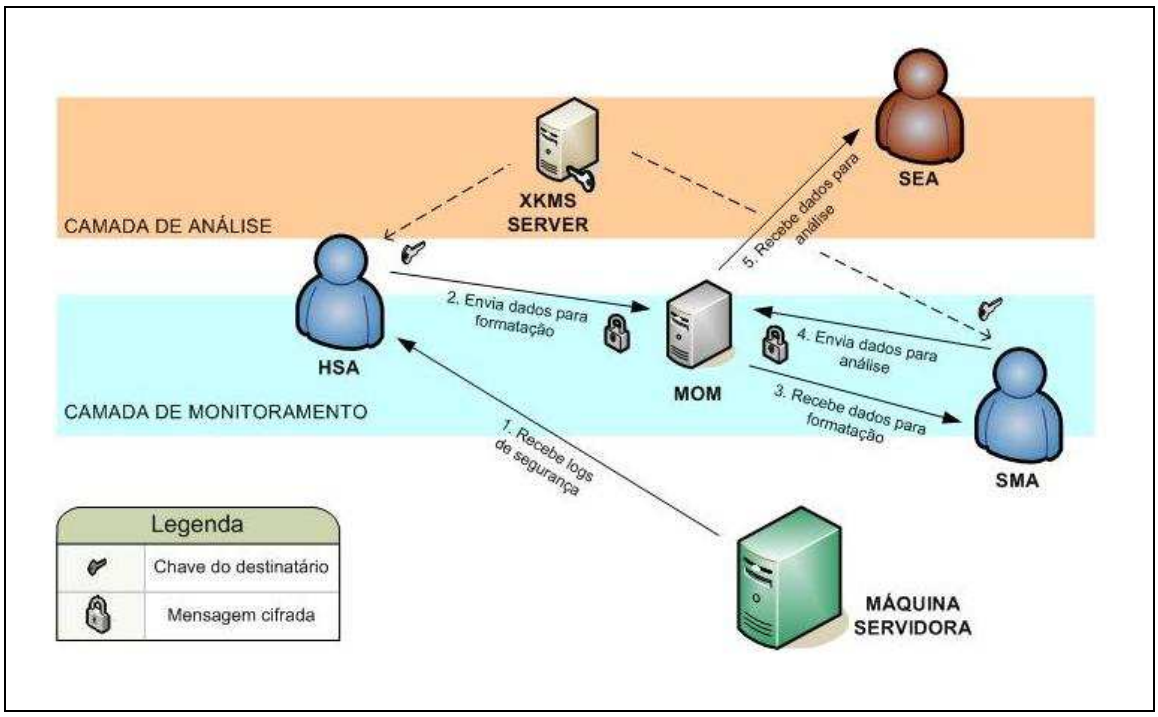

Figura 5.2 – Solução proposta integrada ao processo de coleta de dados do IDS-NIDIA.

De acordo com a Figura 5.2, o agente HSA coleta os dados de *log* do *host*, cifra os dados com a chave pública do agente SMA (fornecida pelo XKMS Server) e envia as informações protegidas para uma fila do MOM destinada às mensagens do agente SMA. Em seguida, o agente SMA captura os dados da fila, decodifica as informações e executa a formatação. Os dados formatados são novamente criptografados pelo agente SMA, utilizando

a chave pública do agente SEA (que também é fornecida pelo XKMS Server) e enviados para outra fila do MOM destinada às mensagens do agente SEA. Finalmente, os dados são coletados da fila e processados pelo agente SEA.

Baseando-se na comunicação segura efetuada na coleta de dados, conforme apresentado da Figura 5.2, percebe-se que as mensagens cifradas pelos agentes sensores pertencentes à camada de monitoramento, utilizam chaves públicas que necessitam ser requisitadas e estar previamente registradas no XKMS Server. Sendo assim, a solução proposta que cria mecanismos seguros para envio de informações, abrange em especial às operações de registro e localização de chaves públicas, executadas pelo XKMS Server. Em outras palavras, a maior parte dos mecanismos de segurança desenvolvidos foi introduzida na segurança do XKMS Server. A Figura 5.3 apresenta uma visão detalhada da solução geral sendo aplicada ao registro de chave pública, executado pelo agente SMA.

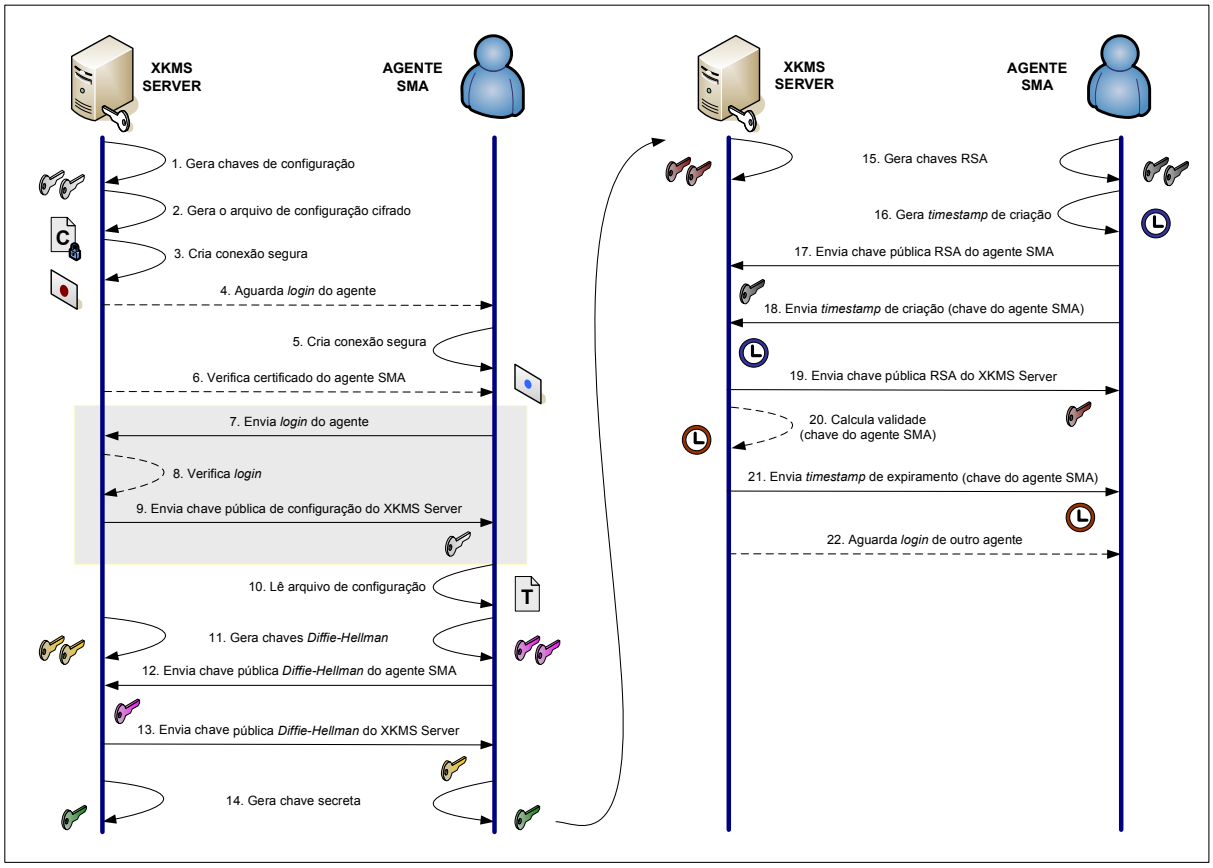

Figura 5.3 – Solução geral aplicada ao registro de chave pública.

Conforme apresentado na Figura 5.3, a operação de registro abrange os mecanismos de proteção de informação, autenticação, autorização e gerenciamento de chaves com *timestamp*. O agente SMA utiliza seu certificado para efetuar a primeira autenticação com o

XKMS Server, a fim de estabelecerem um canal de comunicação seguro. Em seguida, uma segunda autenticação é realizada por meio das informações de *login* do agente SMA, que são enviadas para o XKMS Server (através do canal estabelecido), para que este verifique se o agente que está tentando obter o recurso trata-se de um usuário legítimo. Concluída a segunda etapa de autenticação, a chave pública de acesso ao arquivo de configuração é disponibilizada para o agente e a conexão segura é encerrada. Em posse da chave de configuração, o agente SMA tem acesso aos parâmetros do algoritmo *Diffie-Hellman* (armazenados em arquivo) que são utilizados na geração da chave secreta. Finalmente, a chave secreta (compartilhada entre o agente e o servidor) é utilizada para a troca segura de chaves públicas entre o agente SMA e o XKMS Server e troca de informações (*timestamp*) que irão definir o tempo de vida útil da chave pública registrada no servidor.

A solução geral também se aplica a uma requisição de chave pública ao XKMS Server. Sendo assim, a requisição de chave pública associada ao envio de uma mensagem segura, abrange os mecanismos de autenticação, autorização, gerenciamento de chaves com *timestamp* e persistência de mensagens. A Figura 5.4 ilustra o processo de localização de chave pública e envio de mensagem segura sendo aplicada a dois agentes de diferentes camadas.

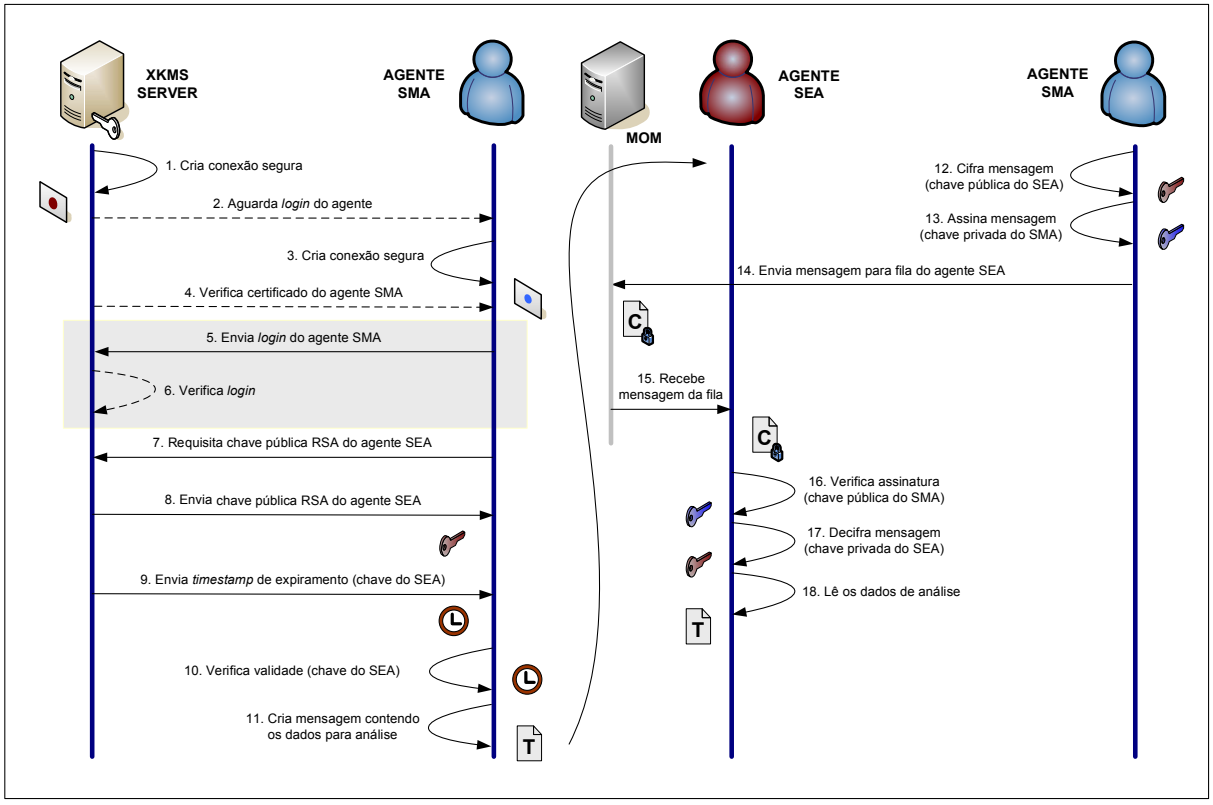

Figura 5.4 – Solução geral aplicada à localização de chave e envio de mensagem segura.

Na Figura 5.4, o agente SMA solicita ao XKMS Server a chave pública do agente SEA para envio de uma mensagem segura contendo *logs* de informação do *host*. Para isso, ele fornece sua credencial de autenticação (certificado SMA) ao XKMS Server e ambos estabelecem uma conexão segura. Em seguida, as informações de *login* do agente SMA são enviadas para o servidor através do canal seguro, para que o mesmo efetue a segunda autenticação. Sendo um usuário legítimo, o agente SMA requisita a chave pública do agente SEA ao XKMS Server e este retorna a chave solicitada e o *timestamp* de expiramento. Em posse da chave, o agente SMA executa uma verificação do *timestamp* para ver se corresponde a uma chave pública válida. Caso a chave pública do agente SEA seja válida, a mensagem é cifrada com essa chave, assinada com a chave privada do agente SMA e enviada para a fila do MOM. Finalmente, o agente SEA captura a mensagem da fila do MOM, verifica se foi assinada pelo agente SMA (através da chave pública desse agente), decifra a mensagem com sua chave privada e processa os dados recebidos.

5.2 Testes da Solução com o IDS-NIDIA

Para proceder com os testes e análise dos resultados obtidos com a solução proposta aplicada ao IDS-NIDIA, algumas ferramentas foram utilizadas para a visualização do comportamento dos agentes. Dentre as ferramentas, pode-se citar:

- O ambiente de desenvolvimento Java Eclipse;
- O agente *Sniffer* (rastreador de mensagens) da interface gráfica do JADE (*Java Agent DEvelopment Framework*);
- A interface gráfica do administrador do OpenJMS, para verificação do comportamento das mensagens na fila;
- O software de rastreamento de pacotes (*Sniffer* de pacotes), denominado Wireshark, para captura e análise dos pacotes que trafegaram na rede durante a execução do protótipo.

Para execução do protótipo, as seguintes variáveis de ambiente foram criadas para acesso às bibliotecas de classes utilizadas na solução:

WASP_HOME = C:\systinet\server_java60\lib
JAAS_HOME = C:\jaas\lib
JMS_HOME = C:\jms1.1\lib
OPENJMS_HOME = C:\openjms-0.7.7-beta-1\lib

Em seguida, as seguintes entradas foram adicionadas à variável de ambiente *CLASSPATH* :

%JAAS_HOME%\lib\jaas.jar;
%JMS_HOME%\lib\javax.jms.jar;
%OPENJMS_HOME%\lib\openjms-0.7.7-beta-1.jar;
%WASP_HOME%\lib\wasp.jar;
%WASP_HOME%\lib\ws_rm_client.jar;

No protótipo, a execução dos agentes e do XKMS Server é acionada através do JADE (*jade.Boot*), sendo que algumas propriedades de configuração foram adicionadas a linha de comando, conforme descritas abaixo:

-Dwasp_home=%WASP_HOME%;
-Djava.security.auth.login.config=="C:\ [caminho] \sample_jaas.config";
-Djavax.net.ssl.trustStore="C:\ [caminho] \keystore"

A entrada *-Dwasp_home* é utilizada no protótipo para estender o MTP (*Message Transport Protocol*) do JADE, a fim de fornecer transporte de mensagens utilizando *web services*. Com o propósito de reforçar a confiabilidade na entrega de mensagens dos agentes, o modelo de MTP desenvolvido por [OLIVEIRA, 2006] foi estendido e adaptado para se integrar com a proposta de mensagem persistente. Sendo assim, a seguinte linha de comando também foi informada:

-mtp prototype.reliability.MessageTransportProtocol

As entradas *-Djava.security.auth.login.config* e *-Djavax.net.ssl.trustStore* são utilizadas respectivamente, para configurar o verificador de *login*, usado na autenticação de usuário, e configurar a fonte de um certificado confiável.

Assim, têm-se as seguintes linhas de comando necessárias para a execução do agente e do XKMS Server, conforme é mostrado, respectivamente nas Listagens 5.1 e 5.2.

Listagem 5.1 – Linha de execução do agente.

```
Java -Dwasp_home=%WASP_HOME% -Djava.security.auth.login.config==
"C:\ [caminho] \sample_jaas.config"; -Djavax.net.ssl.trustStore="C:\ [caminho] \kserver"
jade.Boot -port 1098 -gui -mtp prototype.reliability.MessageTransportProtocol
agent:prototype.samples.Agent
```

Listagem 5.2 – Linha de execução do XKMS Server.

```
Java -Dwasp_home=%WASP_HOME% -Djava.security.auth.login.config==
"C:\ [caminho] \sample_jaas.config"; -Djavax.net.ssl.trustStore="C:\ [caminho] \kclient"
jade.Boot -port 1098 -gui -mtp prototype.reliability.MessageTransportProtocol
xkmsServer:prototype.security.xkmsServer.XKMSServer
```

Na primeira fase de execução do XKMS Server, o servidor cria uma conexão segura em seu ambiente, utilizando seu certificado, e aguarda a conexão de um agente. Quando o agente é executado, este cria uma conexão segura e se comunica com o XKMS Server, habilitando um canal de comunicação através da validação do seu certificado. Este canal seguro é usado para enviar as informações de *login*, oferecendo suporte ao mecanismo de autenticação e controle de acesso. A Figura 5.5 mostra na seqüência: o canal de comunicação seguro sendo criado, as informações de *login* sendo enviadas pelo agente, a autenticação bem sucedida e o acesso autorizado ao recurso do XKMS Server .

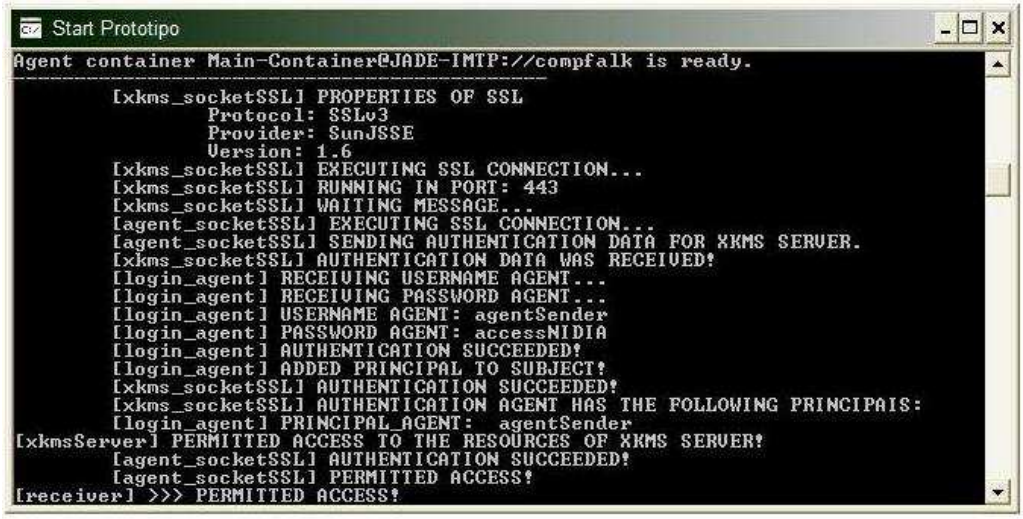

Figura 5.5 – Processo de conexão segura com autenticação do agente.

A Figura 5.6 apresenta a seqüência de mensagens trocadas entre dois agentes (*sender* e *receiver*) e o XKMS Server após uma autenticação bem sucedida. A seqüência mostra duas operações de registro de chaves no XKMS Server e uma operação de requisição de chave no servidor para envio de mensagem segura.

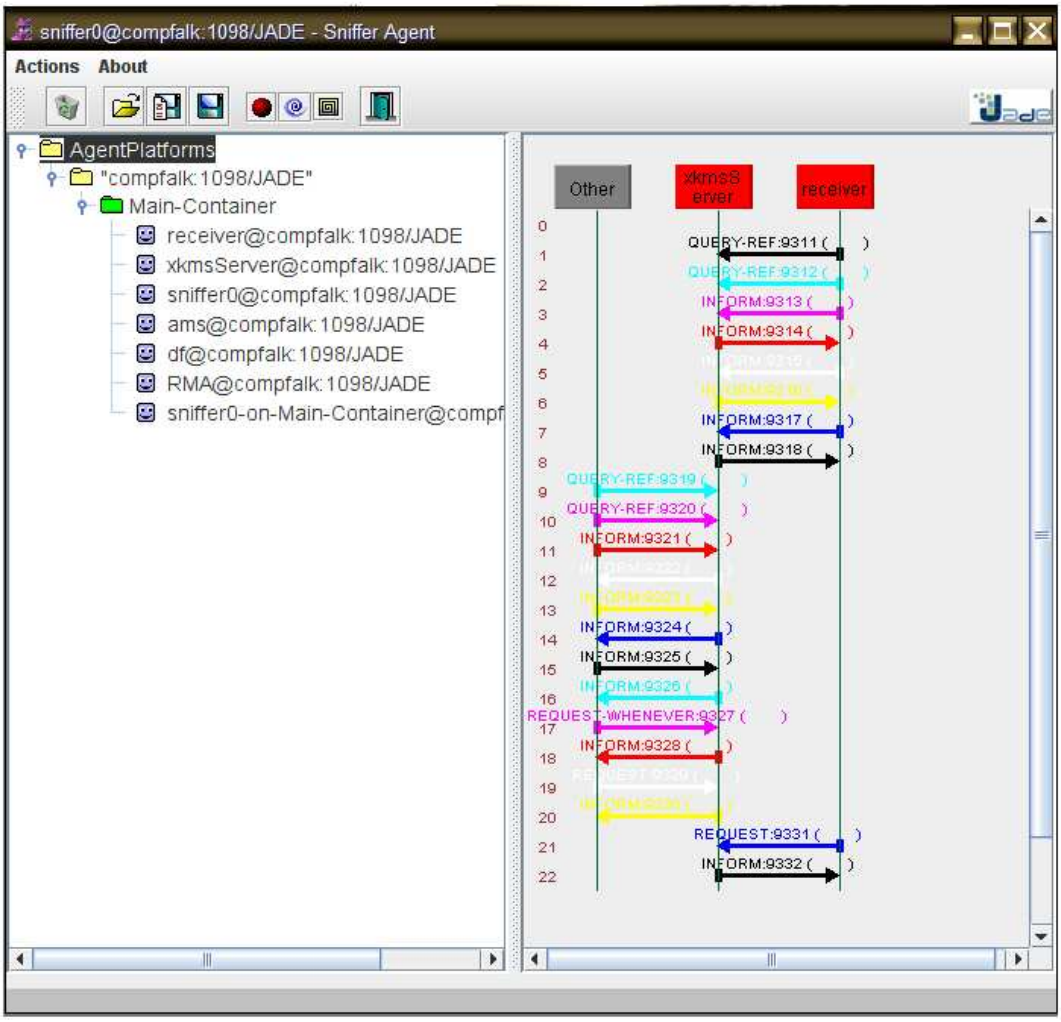

Figura 5.6 – Seqüência de troca de mensagens de registro e localização de chaves.

As mensagens identificadas na Figura 5.6 estão associadas às seguintes operações:

- ***Mensagens QUERY-REF 9311, 9312, 9319 e 9320:*** Requisição de chave pública de configuração;
- ***Mensagens INFORM 9313, 9314, 9321 e 9322:*** Troca de chaves *Diffie-Hellman*;
- ***Mensagens INFORM 9315, 9316, 9323, e 9324:*** Troca de chaves públicas RSA;

- ***Mensagens INFORM 9317 e 9325:*** Envio de *timestamp* de criação da chave pública;
- ***Mensagens INFORM 9318 e 9326:*** Recebimento de *timestamp* de expiramento da chave pública registrada;
- ***Mensagens REQUEST 9329 e 9331:*** Requisição de chave pública;
- ***Mensagens INFORM 9330 e 9332:*** Recebimento de chave pública;
- ***Mensagens INFORM 9328:*** Recebimento do *timestamp* de expiramento da chave pública do agente *receiver*.

As mensagens de requisição de chave pública de configuração (*QUERY-REF*) dão suporte ao mecanismo de proteção de informações de configuração. Sendo assim, elas servem para comunicar ao XKMS Server de que o canal de comunicação segura será utilizado para o recebimento da chave pública de configuração. Nesse caso, o XKMS Server cria uma conexão segura no lado servidor e este aguarda a conexão segura do agente. Quando as duas conexões são criadas, o canal de comunicação seguro é estabelecido e a chave de configuração é remetida para o agente. A Figura 5.7 mostra os resultados dessa operação.

Figura 5.7 – Processo de conexão segura para envio da chave de configuração.

As mensagens de *timestamp* (*INFORM*) dão suporte ao mecanismo de gerenciamento de chaves e são utilizadas na definição do tempo de vida útil das chaves públicas estocadas no XKMS Server. A Figura 5.8 mostra a data de criação da chave do proprietário (*receiver*) sendo recebida pelo XKMS Server e a data de expiramento calculada sendo enviada para o proprietário da chave.

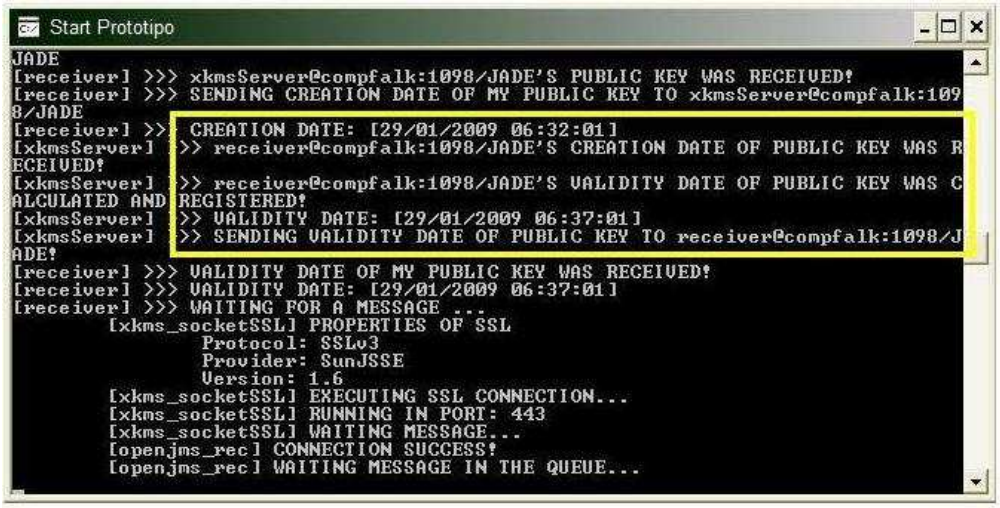

Figura 5.8 – Mensagens de cálculo de *timestamp* de expiramento.

As mensagens de *timestamps* detêm os mesmos mecanismos de proteção utilizados nas mensagens de troca de chaves públicas. Como elas também são utilizadas pelo XKMS Server para alertar proprietários sobre quando sua chave deve ser renovada, convém assegurá-las de que elas não podem ser lidas por intrusos. A Listagem 5.3 mostra como essas mensagens são transmitidas pelos agentes e pelo XKMS Server.

Listagem 5.3 – Mensagem XML com *timestamp* criptografado.

```
1  <?xml version="1.0"?>
2  <rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/TR/1999/PR-rdf-schema-19990303#"
  xmlns:fipa-rdf="http://www.fipa.org/schemas/FIPA-RDF#">
3     <rdf:object>
4        <fipa-rdf:CONTENT_ELEMENT>
5           <rdf:Description>
6              <rdf:type>null#TIME_OF</rdf:type>
7              <TIMEKEY><rdf:Description>
8                 <TIMESTAMP>iCcUSiMv2PViadBdABO0r+8ZGtAL7nh1</TIMESTAMP>
9              </rdf:Description></TIMEKEY>
10             <HOLDER>
11                <rdf:Description>
12                   <NAME>receiver@Falkner-PC:1098/JADE</NAME>
13                </rdf:Description>
14             </HOLDER>
15          </rdf:Description>
16       </fipa-rdf:CONTENT_ELEMENT>
17    </rdf:object>
18 </rdf:RDF>
```

O mecanismo de persistência de mensagens define o último mecanismo utilizado na execução do protótipo. Sendo assim, as operações se resumem no tratamento das mensagens que serão enviadas para a fila do MOM e no tratamento das mensagens que serão recebidas da fila do MOM. Essas operações podem ser visualizadas através do console administrador do MOM, conforme é apresentado na Figura 5.9.

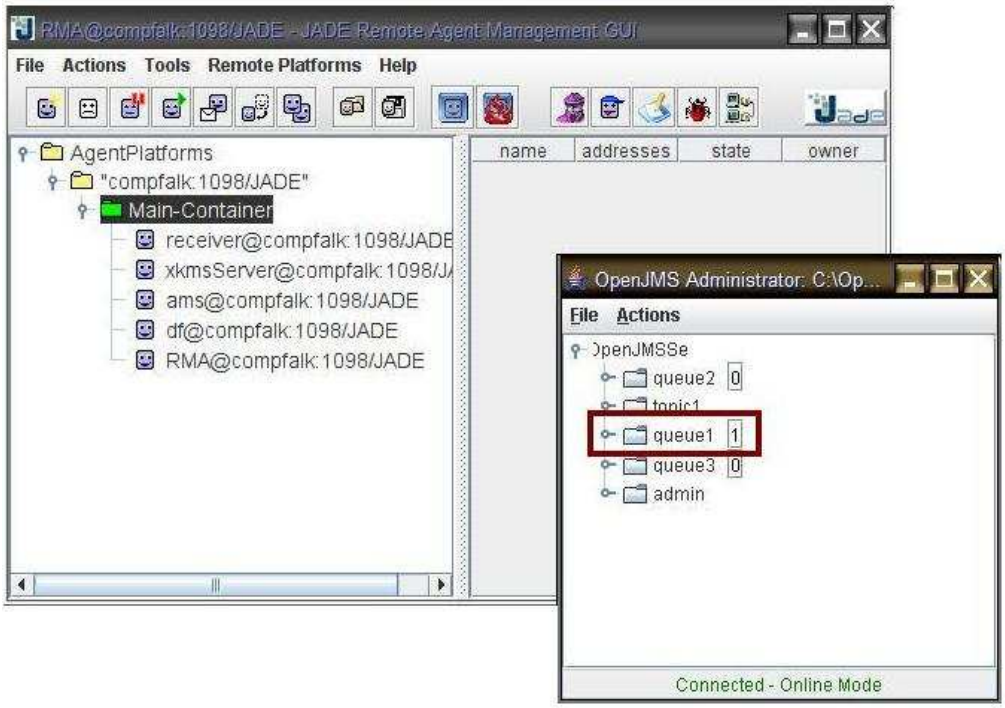

Figura 5.9 – Representação da mensagem armazenada na fila do MOM.

A Figura 5.9 mostra uma mensagem sendo armazenada na fila do OpenJMS (MOM), sendo que esta fila é configurada para o agente *receiver@complfalk:1099/JADE*. Na Figura 5.10, tem-se o agente *receiver@complfalk:1099/JADE* recebendo a mensagem da fila.

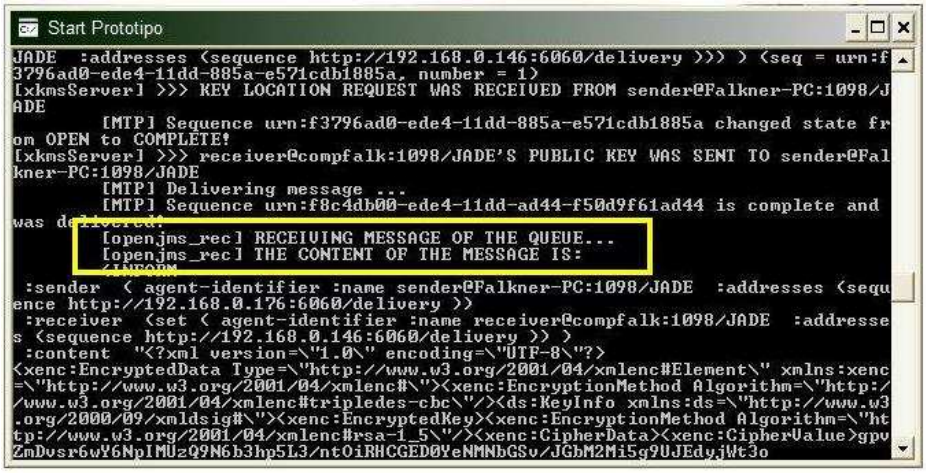

Figura 5.10 – Mensagem recebida da fila pelo agente.

Para testar a funcionalidade do protótipo, utilizou-se um rastreador de pacotes para avaliar a segurança das mensagens trocadas pelos agentes e os canais de segurança criados para envio de informações sensíveis. Esta ferramenta também foi utilizada para constatar o uso correto da porta destinada às mensagens persistentes. Portanto, na Figura 5.11 tem-se a representação dos pacotes de mensagens trocados para a realização do canal seguro.

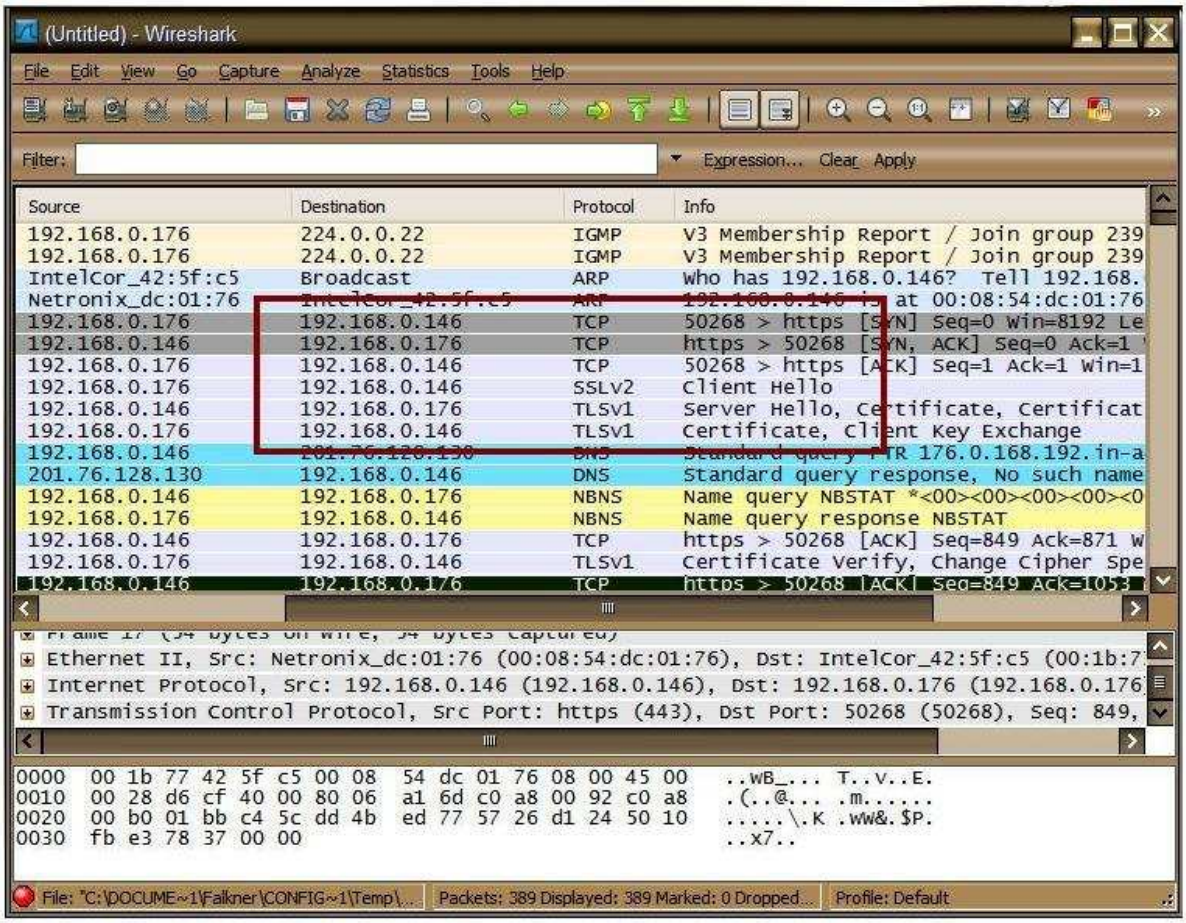

Figura 5.11 – Pacotes de mensagens utilizados na criação do canal seguro.

Na Figura 5.11, o XKMS Server está identificado pelo IP 192.168.0.146 e se comunica com o agente utilizando os protocolos TCP (*Transmission Control Protocol*), SSLv2 (*Secure Sockets Layer – version 2*) e TLSv1 (*Transport Layer Security – version 1*), e a porta 443. O agente está identificado pelo IP 192.198.0.176 e se comunica com o XKMS Server utilizando os mesmos protocolos utilizados no servidor e a porta 50268.

A Figura 5.12 mostra o certificado do agente sendo requisitado pelo XKMS Server como parâmetro de criação do canal seguro.

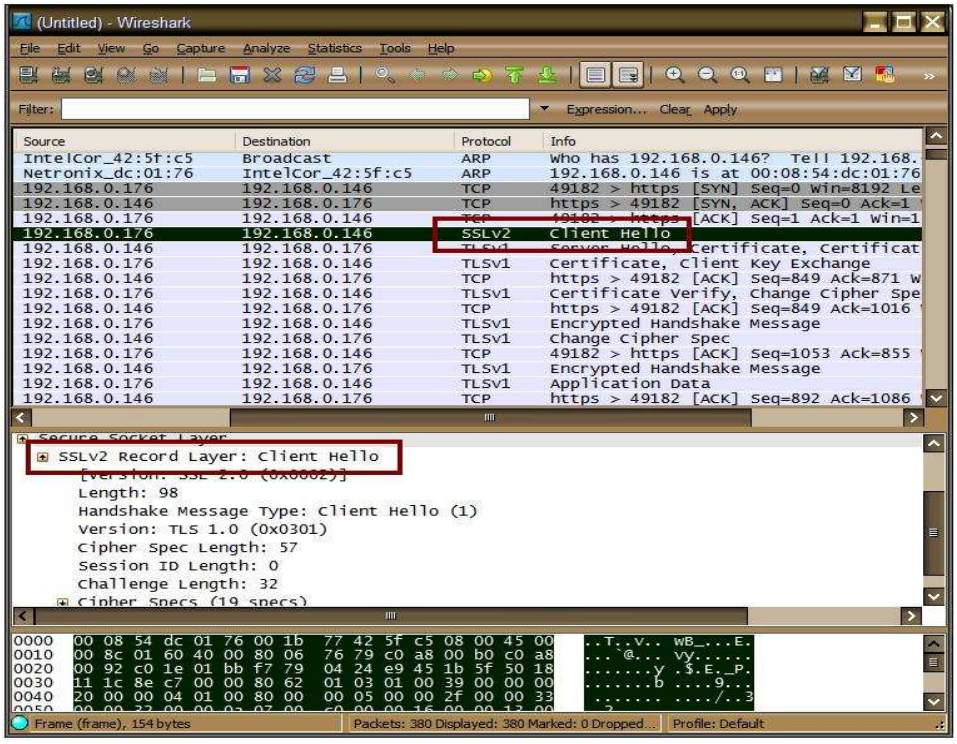

Figura 5.12 – Certificado do agente enviado como parâmetro de criação canal.

A Figura 5.13 mostra a porta 1099 RMI (*Remote Method Invocation*) de comunicação com o MOM, sendo utilizada pelo agente emissor para armazenamento de mensagens.

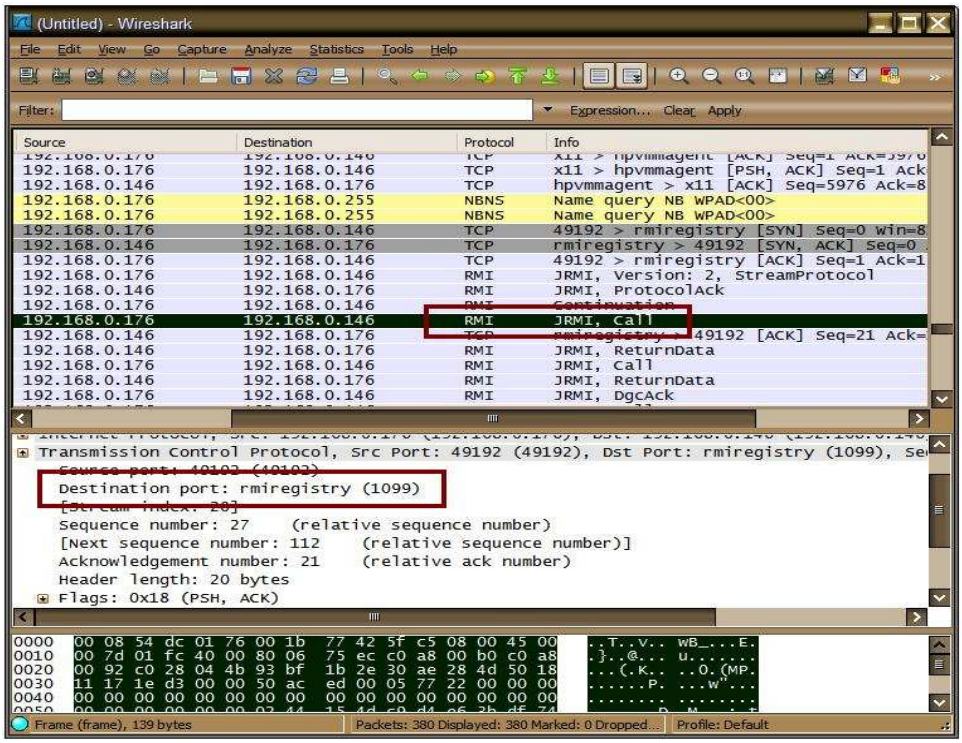

Figura 5.13 – Porta do MOM utilizada para envio de mensagens.

O protótipo implementado teve seus testes realizados nos laboratórios da UFMA, onde se encontra atualmente instalado e em execução.

5.3 Conclusão

Este capítulo apresentou a aplicabilidade da solução inserida no contexto da execução e da comunicação dos agentes do IDS-NIDIA. Nesse sentido, procurou-se demonstrar de uma forma mais abrangente, como as funcionalidades dos mecanismos propostos podem, em conjunto, elevar o patamar de segurança e confiabilidade necessários para o funcionamento estável e seguro dos agentes do IDS-NIDIA. Neste capítulo, os procedimentos necessários para execução do protótipo da solução e o comportamento dos mecanismos de segurança a cada fase de execução também foram apresentados. Os testes do protótipo, que simulam a comunicação de agentes e comprovam a viabilidade da solução, foram apresentados através de ilustrações.

6 Conclusão Geral

Neste capítulo, as conclusões são apresentadas através dos principais pontos alcançados com esta dissertação e suas contribuições. Sugestões de futuras pesquisas na área de segurança de agentes também são delineadas.

6.1 Contribuições do Trabalho

No universo da segurança da informação digital, sistemas de detecção de intrusão (IDS) têm se mostrado grandes aliados como ferramentas robustas de prevenção de ataques em ambientes de rede heterogêneos e de alto fluxo de tráfego. Tornaram-se, portanto, um dos principais temas de investigação e muitas pesquisas têm sido feitas com o intuito de integrar a tecnologia de agentes no desenvolvimento desses sistemas. Devido à capacidade de inteligência e mobilidade, a utilização de agentes de software no projeto de IDS vem proporcionando ganhos significativos em eficiência e escalabilidade, principalmente quando envolve operações de análise de comportamento (buscando atividades de intrusão) e realização de contramedidas (prevenindo novos ataques).

Entretanto, como entidades de software, os agentes podem facilmente ser alvos de ataques externos em ambientes inseguros como a Internet. Em outras palavras, sistemas de detecção que utilizam agentes para garantir a segurança de uma rede distribuída, não necessariamente fazem dos IDS sistemas verdadeiramente seguros de serem utilizados. Assim, a necessidade de garantir a segurança dos próprios componentes desse sistema, durante sua execução e comunicação, transcende a necessidade de proteger a rede do qual eles fazem parte.

Dentro dessa perspectiva, uma alternativa que pudesse viabilizar um sistema de detecção de intrusão que emprega tecnologia de agentes, seria o desenvolvimento de técnicas que pudessem oferecer proteção à execução dos agentes e mecanismos que pudessem dar garantias de integridade e confiabilidade na troca de mensagens entre essas entidades, elevando assim, o conceito de segurança à própria estrutura interna do IDS.

A principal contribuição desse trabalho foi propor uma solução de segurança e confiabilidade para IDS baseados em agentes. Esta solução integrou ao IDS-NIDIA mecanismos de segurança para oferecer proteção de informação de configuração do servidor de chaves, autenticação e controle de acesso aos recursos do servidor, e esquema de

gerenciamento de chaves com uso de *timestamp*. A solução também integrou mecanismos para oferecer confiabilidade e integridade às mensagens dos agentes através de persistência.

Como contribuição às informações sensíveis do IDS, a solução utilizou técnicas de criptografia para garantir proteção desses dados, os quais são compartilhados pelos agentes para o registro de suas chaves públicas no servidor de chaves do IDS. Essas informações, que representam dados confidenciais do servidor e eram disponibilizadas no sistema através de um documento XML legível, passaram a ser geradas, criptografadas e armazenadas dinamicamente pelo servidor.

Para o reconhecimento de forma confiável das entidades envolvidas na comunicação e o controle eficaz do nível de acesso dessas entidades, a solução contribuiu com um esquema de autenticação e controle de acesso aos recursos do servidor de chaves do IDS. Este esquema utilizou duplo serviço de autenticação para acesso aos recursos do servidor. A primeira autenticação baseou-se na verificação do certificado do agente para que um canal de comunicação segura possa ser estabelecido entre o agente e o servidor. Através desse canal, os dados de autenticação do agente são enviados. A segunda autenticação baseou-se nas informações de *login* fornecidas pelo agente para que o recurso de acesso a chave de configuração possa ser liberado para o agente. Este recurso restringe o acesso ao servidor de chaves somente aos agentes do sistema.

No quesito gerenciamento de chaves com *timestamp*, a solução contribuiu com um modelo de gerenciamento mais seguro para o servidor, que utiliza *timestamp* para determinar um tempo de vida útil às chaves públicas armazenadas. Este modelo protege o sistema contra chaves comprometidas, caso elas sejam capturas, analisadas e reutilizadas por indivíduos não autorizados.

A solução contribuiu também com canais de persistência na troca de mensagens, como meios de proteção contra falhas do IDS ou da rede. Este mecanismo utilizou um provedor de mensagens persistentes para assegurar a integridade e a confiabilidade ao IDS. As mensagens dos agentes passaram a trafegar de forma indireta, utilizando filas de mensagens persistentes.

Por fim, a solução buscou corrigir algumas falhas presentes na estrutura interna de IDS executados por agentes. O funcionamento em conjunto dos mecanismos desenvolvidos, minimizaram certas vulnerabilidades presentes no uso da tecnologia de agentes em sistemas de detecção, permitindo garantir viabilidade quanto ao uso dessa ferramenta.

6.2 Considerações Finais

Este trabalho procurou enfatizar a importância no desenvolvimento de mecanismos de defesa que corrijam falhas contidas na estrutura interna de sistemas IDS baseados em agentes, privilegiando a execução e a comunicação segura dos seus componentes, como única forma de torná-los menos suscetível a ameaças que possam comprometer o papel principal desses sistemas que é garantir a segurança da rede.

A solução proposta baseou-se na continuidade do trabalho de [OLIVEIRA, 2006], que propôs um esquema de comunicação segura e confiável para sistemas multiagentes utilizando especificações de segurança XML, e que poderiam ser aplicadas a um sistema de detecção de intrusão baseado em agentes. A solução possuía algumas limitações e precisavam ser supridas para dar mais consistência ao objetivo proposto.

Apesar de todos os esforços para descobrir qual seria o melhor método que pudesse ser empregado à segurança no envio das informações de *login* do agente e recebimento da chave de configuração do servidor, a opção de utilizar SSL não descarta a possibilidade de este mecanismo sofrer um ataque do “homem-do-meio”, pois é certo que esse tipo de protocolo ainda abriga algumas vulnerabilidades que podem ser exploradas pelo invasor. No entanto, esse protocolo foi utilizado no esquema de autenticação e autorização, pois a solução ainda conta com o mecanismo de transferência do arquivo de configuração via canal seguro (SFTP), configurado pelo administrador do IDS-NIDIA. Sendo assim, mesmo que um intruso consiga invadir o canal e capturar a chave de configuração, este terá outro obstáculo de segurança que acessar o arquivo de configuração.

Outro ponto a ser considerado, é a necessidade de definir um tempo mínimo e um tempo máximo, para que a chaves públicas permanecem armazenadas no repositório de chaves do servidor. Um tempo suficientemente pequeno poderia impossibilitar o envio de uma mensagem cifrada, pois a própria rede está sujeita a congestionamentos, fazendo com que a mensagem chegue ao seu destinatário, cifrada com uma chave pública expirada. Em contra partida, um tempo suficientemente grande, poderia expor a chave a processos criptoanálise por força bruta por parte do invasor.

Todos os mecanismos desenvolvidos nesta solução foram testados com sucesso no protótipo da solução apresentado neste trabalho. Para a solução de persistência, dois agentes de software (um emissor e um receptor) foram utilizados e uma fila do provedor foi configurada para os testes serem executados.

6.3 Limitações

Apesar dos pontos positivos alcançados com este trabalho, algumas limitações acabaram sendo identificadas. Dentre elas pode-se citar:

- O canal de comunicação segura utilizado no mecanismo de autenticação e autorização baseia-se no protocolo SSL, sendo que este possui vulnerabilidades e podem ser explorados pelo invasor através do ataque do “homem-do-meio”;
- O XKMS Server não define tempo mínimo e tempo máximo para armazenamento de chaves no repositório;
- O XKMS Server não define validade para a sua chave pública, sendo que ele a utiliza permanentemente até que seja inicializado novamente;
- O XKMS Server não fornece meios para assegurar que o arquivo de configuração encontra-se disponível no seu ambiente, e não fornece mecanismo que possam recriá-los em tempo de execução caso seja apagado ou movido do seu local de origem. A distribuição e a autorização para acesso ao arquivo de configuração é parte das responsabilidades do administrador do IDS-NIDIA.

6.4 Trabalhos Futuros

Para continuidade deste trabalho e sugestões para trabalhos futuros, pode-se citar:

- Desenvolver um mecanismo mais seguro para envio das informações de autenticação do agente;
- Desenvolver um serviço de gerenciamento de chaves distribuído, fazendo com que qualquer agente do sistema possa desempenhar o papel de um servidor de chaves;
- Criar um simulador de ataque para testar a segurança dos agentes de um IDS.

7 Referências Bibliográficas

APACHE. **Apache Modules**, 2008. Disponível em: <http://info.iqm.unicamp.br/manuais /apache/mod/mod_ssl/ssl_reference.html>

BELLIFEMINE, Fabio; CAIRE, Giovanni; GREENWOOD, Dominic. **Developing multi-agent systems with JADE**. Publisher: John Wiley & Sons Ltd, 2007.

BRIFFAUT, Jeremy. **MIDS: Multi-Level and Multi-Agent Intrusion Detection System**. In First International Workshop on Privacy and Security in Agent-based Collaborative Environments, Japan Hakodate, 2006.

DASGUPTA, D.; GONZALEZ, F.; YALLAPU, K.; GOMEZ, J.; YARRAMSETTII, R. **CIDS: An Agent-based Intrusion Detection System**. In Computers & Security, 2005.

DEBAR, Hervé. **Intrusion Detection Systems: Introduction to Intrusion Detection and Analysis, Security and Privacy in Technologies**. Nato Science Series III Vol. 193, IOS Press, edited by Borka Jerman Blazic, Wolfgang Schneider and Tomaz Klobular, 2004.

DIFFIE-HELLMAN. **Diffie-Hellman Key Agreement Method RFC 2631**, 2007. Available in: <http://tools.ietf.org/html/ rfc2631>.

ECHAVARRÍA, Isabel Cristina Satizábal. **Contribución a la Validación de Certificados em Arquitecturas de Autenticación e Autorización**. Tesis (Doctorado. Em Ingenieria Telemática). Universitat Politécnica de Catalunya. Barcelona, 2007.

FIPA. **FIPA Home Page**, 2008. Disponível em <http://www.fipa.org/>. IBM WebSphere MQ. **IBM WebSphere MQ**. Access date: 27/07/2008. Available in: <http://www306.ibm.com/software/integration/wmq/index.html>.

JADE. **JADE Home Page**, 2008. Disponível em: <http://jade.tilab.com/>.

JMS. **JMS Home Page**, 2008. Disponível em: <http:\\java.sun.com/produtcs/jms/>.

LIMA, C. F. L. **Agentes Inteligentes para Detecção de Intrusos em Redes de Computadores**, 2001. 110f Dissertação (Mestrado. em Engenharia de Eletricidade) - Universidade Federal do Maranhão. São Luís, MA, 2001.

LUCENA, M. J. **Criptografía y Seguridad en Computadores**, 2ª Ed., Universidad de Jaén, Jaén, 2003.

MENASCÉ, Daniel A. **MOM vs. RPC: Communication Models for Distributed Applications**. In Internet Computing, IEEE. Vol 9, Issue 2, Page(s): 90-93, 2005.

MENEZES, A. J.; OORSCHOT, P. C. V.; VANSTONE, S. A. **Handbook of Applied Cryptography**. CRC Press Boca Raton, 1997.

NAKAMURA, Emílio Tissato; GEUS, Paulo Licio de. **Segurança de Redes em Ambientes Cooperativos**. Editora Novatec, 488p, 4ª edição, 2007.

NATSCAPE. **Natscape Communications Home Page**, 2008. Disponível em: http://netscape.aol.com/

OLIVEIRA, Emerson J. S. **Comunicação Segura e Confiável para Sistemas Multiagentes Adaptando Especificações XML**. 2006. 114f. Dissertação (Mestrado. em Engenharia de Eletricidade) - Universidade Federal do Maranhão. São Luís, MA, 2006.

OPENJMS. **Openjms Home Page**, 2008. Disponível em: <http://openjms.sourceforge.net/>.

RAMCHURN, Sarvapali D.; HUYNH, Dong; JEN-NING, Nicholas R. **Trust in Multi-Agent Systems**. In Cambridge University Press. New York, USA, 2005.

RASHID, Awais; CHITCHYAN, Ruzanna. **Persistence as an Aspect**. Proceedings of the 2nd International Conference on Aspect-oriented Software Development. Boston, Massachusetts. Pages: 120-129, 2003.

REHAK, Martin; PECHOUCEK, Michal; BARTOS, Karel; GRILL, Martin; CELEDA, Pavel; KRMICEK, Vojtech. **CAMNEP: An Intrusion Detection System for High-Speed Networks**. In Progress in Informatics, (JPN), ISSN 1349-8614, Vol. 2008, n-5, pp. 65-74.

SILVA, Alberto. **Agentes de Software na Internet**. Ed. Centro Atlântico, p-196, 1ª edição, 1999.

SILVA, M. L. Carvalho, **Modelo de IDS Remoto Baseado na Tecnologia de Agentes, Web Services e MDA**. 2006. 142f. Dissertação (Mestrado. em Engenharia de Eletricidade) - Universidade Federal do Maranhão. São Luís, MA, 2006.

SNORT. **Snort Home Page**, 2009. Disponível em> http://www.snort.org/.

STANIFORD-CHEN, S. **Common Intrusion Detection Framework (CIDF)**. Computer Emergency Response Team (Coordenation Center), Outubro 1998. Disponível em: <http://gost.isi.edu/cidf/>.

STELL, A. J.; SINNOTT, R. O.; WATT, J. P. **Comparison of Advanced Authorization Infrastructures for Grid Computing**. 19th International Symposium on High Performing Computing Systems and Applications (HPCS'05), pp. 195-201, 2005.

SYSTINET. **Systinet Home Page**, 2006. Disponível em: <http://www.systinet.com/>.

TIBCO Rendezvous. **TIBCO Rendezvous**. Access date: 27/07/2008. Available in: <http://www.tibco.com/software/messaging/rendezvous/default.jsp>.

VAN TILBORG, C. **Encyclopedia of Cryptography & Security**. Springer, 2005.

VERISIGN. **VeriSign Home Page**, 2008. Disponível em: <http://www.verisign.com/>.

W3C. **W3C Home Page, 2008**. Disponível em: http://www.w3.org\

WOUTERS, Karel; PRENEEL, Bart. **Towards an XML Format for Time-Stamps**. ACM Workshop on XML Security, Nov. 22, 2002, Fairfax VA, USA, 2002.