Classes para Comunicação de Rede com Criptografia e Autenticação usando Java e CORBA

Luiz Angelo Barchet Estefanel
lae@inf.ufsm.br

UNIVERSIDADE FEDERAL DE SANTA MARIA

CENTRO DE TECNOLOGIA

CURSO DE BACHARELADO EM INFORMÁTICA

CsocketS - CLASSES PARA COMUNICAÇÃO DE REDE COM ENCRIPTAÇÃO E AUTENTICAÇÃO USANDO JAVA E CORBA

Trabalho de Graduação Nº 57

Luiz Angelo Barchet Estefanel

Santa Maria, 3 de abril de 1999

UNIVERSIDADE FEDERAL DE SANTA MARIA

CENTRO DE TECNOLOGIA

CURSO DE BACHARELADO EM INFORMÁTICA

A Comissão Examinadora, abaixo-assinada, aprova a monografia intitulada "**CsocketS – Classes para Comunicação de Rede com Criptografia e Autenticação usando Java e CORBA**", elaborada por **Luiz Angelo Barchet Estefanel** como requisito parcial para a obtenção do grau de Bacharel em Informática.

Comissão Examinadora: ____

João Carlos Damasceno Lima - Orientador

____

Iara Augustin

____

Marcelo Pasin

Santa Maria, 3 de abril de 1999.

AGRADECIMENTOS

Ao Professor João Carlos Damasceno Lima, por sua orientação e constante busca por aplicações práticas sobre o que foi desenvolvido.

Aos colegas do Departamento de Suporte do Centro de Processamento de Dados da UFSM, que tiveram paciência e colaboração quando o desenvolvimento do sistema tomava grande parte dos meus esforços.

Ao Diretor do Centro de Processamento de Dados da UFSM, Sérgio Limberger, e ao Professor Cláudio Rocha Lobato, sempre interessados nas possibilidades de desenvolvimento oferecidas através desse trabalho.

Ao Curso de Bacharelado em Informática e à Universidade Federal de Santa Maria, pelo apoio no meu aperfeiçoamento profissional.

Aos professores do curso, por seu estímulo, carinho, conhecimento e interesse.

À minha namorada, Manuele, pelo constante estímulo ao desenvolvimento do sistema, pelo questionamento necessário frente aos problemas encontrados, pela revisão do trabalho e pelo carinho e apoio nos momentos de incerteza. Sem ela, não teria conseguido terminar esse trabalho em tempo hábil.

À minha família, Valduino, meu pai, Ilda, minha mãe, e Felipe, meu irmão, que sempre confiaram e estimularam, trazendo ânimo nos momentos de dificuldade e alegria para seguir em frente.

A todos que de alguma forma auxiliaram, agradeço.

O autor

SUMÁRIO

AGRADECIMENTOS vii

SUMÁRIO v

RESUMO vii

ABSTRACT viii

LISTA DE TABELAS ix

LISTA DE FIGURAS x

LISTA DE ANEXOS xi

LISTA DE ABREVIATURAS xii

1 INTRODUÇÃO 1

1.1 MOTIVAÇÃO 1
1.2 OBJETIVOS 1
1.3 ORGANIZAÇÃO DO TRABALHO 3

2 REVISÃO TEÓRICA 6

2.1 CRIPTOGRAFIA 6
2.1.1 Operações de Cifragem 7
2.1.2 Criptoanálise 9
2.1.3 Chaves Secretas e Públicas 10
2.1.4 Algoritmos de Chave Única ou Secreta (Simétricos) 11
2.1.5 Algoritmos de Chave Pública (Assimétricos) 13
2.2 AUTENTICAÇÃO 16
2.2.1 Conceitos 17
2.3 A LINGUAGEM JAVA 20
2.4 SOCKETS 22
2.5 CORBA 23

3 ANÁLISE DO SISTEMA 26

3.1 ANÁLISE GERAL 27
3.2 TICKETS 37
3.3 SERIALIZAÇÃO 40
3.4 PROCESSOS CRIPTOGRÁFICOS – ASSINATURA DIGITAL E CIFRAGEM 43
3.5 ANÁLISE DE INTERFACES 47
3.5.1 Interfaces Socket 48
3.5.2 Operações de Segurança 50
3.5.3 Factory Server 50
3.5.4 Mapeamento das interfaces IDL 51

4 MODELAGEM DA ARQUITETURA 55

4.1 INTRODUÇÃO 55
4.2 DEFINIÇÃO DAS ESTRUTURAS CORBA ATRAVÉS DA IDL 55

4.3 ADICIONANDO PRIVACIDADE À MENSAGEM 58
4.3.1 Tickets *58*
4.3.2 Serialização *59*
4.3.3 Criptografia *59*
4.3.4 A classe Packer, coordenadora da transformação *61*
4.4 CLIENTES E SERVIDORES 62
5 APLICAÇÕES DESENVOLVIDAS 64
5.1 A ESCOLHA DAS APLICAÇÕES 64
5.2 A APLICAÇÃO TN3270 JAVA 65
5.3 A APLICAÇÃO TELNET 68
5.4 OBSERVAÇÕES FINAIS 71
6 CONCLUSÃO 73
6.1 TRABALHOS FUTUROS 75
7 BIBLIOGRAFIA 76
ANEXOS 78
ANEXO I – Documentação das Classes Desenvolvidas 79
ANEXO II – Documentação da Aplicação TN3270 93
ANEXO III – Documentação da Aplicação Telnet 98
ANEXO IV – Documentação do Pacote Cryptonite 107

RESUMO

A segurança é um requisito fundamental para os sistemas desenvolvidos para a Internet. Aplicações mais antigas, como os terminais *TN3270* e *telnet* requerem atualizações normalmente muito dispendiosas. O conjunto de classes para a comunicação de rede com autenticação e criptografia _ CsocketS _ tem por objetivo oferecer um sistema de fácil integração ao código das aplicações já existentes, estendendo as funcionalidades dos *sockets* para uma comunicação segura, através do emprego de chaves criptográficas RSA para a cifragem e a assinatura digital, e *tickets* de sessão para a autenticação. O sistema foi desenvolvido em Java, sobre a arquitetura de objetos distribuídos CORBA, garantindo facilidade de distribuição e padronização da interface. A arquitetura cliente-servidor também possibilita a implementação do servidor como redirecionador de conexões, possibilitando o tratamento de aplicações diversas por um único servidor. Isso também torna o sistema apto a operar através de *firewalls*. A utilização das classes na extensão de duas aplicações (*TN3270* e *telnet*) demonstrou que o sistema é eficaz para esses sistemas, garantindo níveis de segurança comparáveis a outros sistemas como *Kerberos* e *ssh*, os quais exigem mudanças severas à estrutura das aplicações que os utilizam, o que não ocorre com o uso das classes CsocketS.

ABSTRACT

Security is a basic requiriment for the systems developed to the Internet. Older applications, like the TN3270 and telnet terminals require updates that usually are expensives. The set of classes for network communications with authentication and encryption _ CsocketS _ have as goal to offer a system that could be easily integrated to the programming code from existing applications, extending the funcionalities from the sockets to a safe communication, through the use of RSA cryptographic keys for the encryption and digital signature, and session tickets for the autentication. The system was developed using Java, with the distributed object's architecture CORBA, guaranteeing easiness of distribution and the interface's standardization. The client-server architecture also makes possible the implementation of the server as a connection's relocator, enabling the handling to different applications with the use of one same server. This also enables the system to work through firewalls. The use of the CsocketS classes to the extension of two applications (Tn3270 and telnet) shows that the system if efficient for these systems, providing comparable levels of security to other systems like Kerberos and ssh, which demand several changes to the structure of the applications that use them, what doesn't happen with the use of CsocketS classes.

LISTA DE TABELAS

Tabela 1 _ Alguns Sistemas de Chave Única 12
Tabela 2 _ Alguns Sistemas de Chave Pública.. 15
Tabela 3 _ Alguns Sistemas de *Hash*... 19
Tabela 4 _ Tempo de geração de chaves RSA.. 44
Tabela 5 _ Tempos de Cifragem e Decifragem 45
Tabela 6 _ Operações sobre as Chaves 46

LISTA DE FIGURAS

Figura 1 _ Estrutura de Comunicação........ 22
Figura 2 _ Arquitetura de Referência OMA [VOG 97a] 24
Figura 3 _ Interfaces CORBA [VOG 97a] 24
Figura 4 _ Conexões de um Applet Java 29
Figura 13 _ Segurança Através de *Firewall* 30
Figura 14 _ Seqüência de Mensagens entre Cliente-Servidor-Serviço........ 31
Figura 15 _ Aspectos da Comunicação Cifrada........ 33
Figura 16 _ Fluxo do Processamento da Mensagem 35
Figura 17 _ Camadas de Segurança sobre a Mensagem........ 36
Figura 18 _ Encapsulamento dos Tipos Primitivos 38
Figura 19 _ Estrutura Interna de um *Ticket* 39
Figura 20 _ Serialização de um Objeto........ 41
Figura 21 _ Serialização de um Objeto Fusca 42
Figura 22 _ Operações de Criptografia........ 47
Figura 23 _ Definição dos Servidores........ 51
Figura 24 _ Declaração de um *Array* de *bytes*........ 52
Figura 25 _ Declaração das Chamadas a CSServer........ 53
Figura 18 _ Declaração das chamadas a ConnFactory 54
Figura 19 _ Ligação Cliente/Servidor Através do ORB 56
Figura 20 _ Designação de Servidores pelo *Factory Server*........ 57
Figura 21 _ Classe *TicketUtil*........ 59
Figura 22 _ As Classes Criptográficas do Sistema Cifrando uma Mensagem 60
Figura 23 _ Processos de Cifragem sobre os Dados Enviados........ 61
Figura 24 _ CSServer Atuando no Redirecionamento........ 61
Figura 25 _ Hierarquia de Classes do Cliente........ 62
Figura 26 _ Hierarquia de Classes do Servidor 63
Figura 27 _ Estrutura do *Applet Tn3270*........ 66
Figura 28 _ Substituição da Referência à Classe *Socket*........ 67
Figura 29 _ Tela de Abertura de uma Sessão TN3270 68
Figura 30 _ Parte da Estrutura do *Applet Telnet*........ 69
Figura 31 _ Falha na Identificação dos *Tickets*........ 70
Figura 32 _ Seção de *telnet* Aberta com o *Applet*........ 71

LISTA DE ANEXOS

ANEXO I – Documentação das Classes Desenvolvidas 79
ANEXO II – Documentação da Aplicação TN3270... 93
ANEXO III – Documentação da Aplicação Telnet 98
ANEXO IV – Documentação do Pacote Cryptonite... 107

LISTA DE ABREVIATURAS

API	*Application Program Interface*
CORBA	*Common Object Request Broker Architecture*
DES	*Data Encryption Standard*
GPL	*General Public License*
HTML	*Hypertext Markup Language*
ICMP	*Internet Control Message Protocol*
IDL	*Interface Definition Language*
IIOP	*Internet Inter-ORB Protocol*
IOR	*Interoperable Object Reference*
IPC	*Unix-to-Unix Interprocess Communications*
JDBC	*Java Database Connection*
JDK	*Java Development Kit*
JIT	*Just-in-Time Compiler*
JVM	*Java Virtual Machine*
NFS	*Network Filesystem*
ODBC	*Open Database Connection*
OMG	*Open Management Group*
ORB	*Object Request Broker*
POP-3	*Post Office Protocol version 3*
RMI	*Remote Method Invocation*
RPC	*Remote Process Call*
RSA	*Sistema criptográfico criado por Ron **R**ivest, Adi **S**hamir e Leonard **A**delman*
SSH	*Secure Shell*
SSL	*Secure Socket Layer*
TCP	*Transmission Control Protocol*
TCP/IP	*Transmission Control Protocol/Internet Protocol*
UDP	*User Datagram Protocol*
XDR	*External Data Representation*

1 INTRODUÇÃO

1.1 Motivação

A popularização da Internet possibilitou a difusão do modelo cliente-servidor entre os sistemas computacionais, tornando o uso desses sistemas muito mais flexível e eficiente.

Infelizmente, essa popularização também trouxe alguns problemas, principalmente os relacionados à segurança da rede e dos dados, levando à necessidade de implementar sistemas de segurança, muitas vezes envolvendo o uso de *firewalls* e servidores *proxy* para atuar como uma barreira protetora da rede interna. A necessidade de sigilo para os dados trafegados também é um ponto essencial, pois a divulgação da senha de um usuário, por exemplo, põe em risco todo o sistema de segurança.

Entretanto, não é válido aumentar a segurança sem pensar antes nas aplicações já existentes, e como elas irão se comportar frente à esse novo ambiente. Especialmente os antigos sistemas que utilizam o acesso baseado em terminais, através de serviços tradicionais de rede (*telnet*, *TN3270*, etc.), não estão adaptados a essas exigências. Quando é necessário o acesso externo a essas aplicações, e as medidas de segurança barram o acesso, novas soluções são requeridas. Rescrever as aplicações, de modo a adaptá-las aos novos desafios de um ambiente cliente-servidor, usualmente acarreta custo e tempo demasiadamente elevados. No pior caso não há uma solução alternativa, se as aplicações desenvolvidas para substituir os antigos sistemas falham nos seus objetivos.

1.2 Objetivos

Este trabalho define e implementa um conjunto de classes de comunicação segura, que possibilita estender as formas de acesso das antigas aplicações, aumentando sua vida útil e possibilitando sua reinserção no conjunto de serviços disponíveis, adequando-se às novas necessidades de segurança da rede.

As classes desenvolvidas não só habilitam o acesso remoto de forma adaptada às novas restrições de segurança, como garantem o sigilo dos dados trafegados em

ambiente externo ao da rede da empresa, adicionando mais um nível de proteção ao sistema. Esse sigilo é obtido através da associação de métodos de criptografia e autenticação, garantindo a privacidade dos dados.

Para desenvolver esse conjunto de classes, foram definidos alguns objetivos específicos, que visam moldar os parâmetros e as situações a que se indica o uso.

O primeiro objetivo visa o desenvolvimento de um sistema que execute operações similares às das conexões *socket* de redes, semelhantes tanto em função quanto em formato. Ao usar um sistema com características similares aos *sockets* [JDK 97], obtém-se um amplo espectro de utilização, uma vez que a programação de redes a nível de *socket* é ao mesmo tempo simples, eficiente e popular. O uso de métodos com mesma função, nomenclatura e parâmetros propicia uma forma rápida de adaptação de aplicações já existentes, com o mínimo de modificações à sua estrutura e operação. Esse objetivo é alcançado desenvolvendo uma arquitetura de comunicação de rede que mantenha os mesmos princípios de um *socket*.

Outro objetivo, visando a segurança, constitui a forma com que os dados serão protegidos. A necessidade de proteger as informações que trafegam na rede é devida tanto ao caráter privativo dos dados operados, onde, por exemplo, a inserção de um valor ilegal acarreta uma modificação nas operações realizadas, quanto à própria segurança do sistema de proteção da rede, pois a descoberta de uma senha pode anular todos os esforços de garantir segurança através de *firewalls*. Dessa forma, o sigilo quanto aos dados será realizado através da aplicação sucessiva de três sistemas, autenticação por *tickets*, assinatura digital e criptografia. Por meio da autenticação por *tickets*, são controlados a ordem das transmissões e o tempo máximo de duração da conexão. Usando chaves criptográficas assimétricas do tipo RSA, são também providos os sistemas de assinatura digital, que é outra forma de autenticação da origem, e a criptografia, que possibilita o conhecimento dos dados somente àqueles quem a mensagem é destinada.

Todos esses sistemas são então integrados a um sistema cliente-servidor que utiliza a arquitetura CORBA, para mediar a transmissão dos dados, e a linguagem Java, para possibilitar a mais ampla distribuição do sistema. Da forma como o sistema foi projetado, no lado do servidor ainda é acrescentada uma forma de redirecionar conexões, adequando as aplicações ao uso através de um *firewall*.

Um objetivo secundário do trabalho é a disponibilização de todo o sistema desenvolvido para estudo e uso. Para isso, somente foram usados sistemas cuja utilização e distribuição é livre ou segue a licença de distribuição GPL (*General Public License* da *Free Software Foundation*[1]), que garante a disponibilidade do código fonte e uso livre.

1.3 Organização do Trabalho

Neste trabalho é incluída a descrição de todo o processo de criação do sistema, os problemas, as soluções e curiosidades encontradas. A documentação dessa experiência é uma ferramenta importante para desenvolvedores que desejam continuar o trabalho ou utilizar parte dos sistemas criados.

No Capítulo 2 é feita uma extensa revisão bibliográfica dos temas pertinentes à segurança de dados, especificamente criptografia e autenticação, e são apresentados o Java [JDK 97], os *Sockets* [ORF 97] e a arquitetura CORBA [VOG 97a], também utilizados no desenvolvimento e suporte do sistema. Essa revisão possibilita esclarecer o funcionamento desses conceitos, bem como propicia informações adicionais que determinam a utilização de métodos específicos, por exemplo na criptografia e na autenticação.

No Capítulo 3 é detalhada a implementação das classes. A Análise Geral conduz através dos motivos que levaram às decisões de implementação, dos problemas encontrados (sobretudo com o CORBA), e das vantagens deste sistema frente aos protocolos de segurança mais conhecidos, mostrando a estrutura de segurança projetada para atuar sobre os dados.

Ainda no Capítulo 3, são detalhados os procedimentos e métodos utilizados para a implementação do sistema. Inicialmente exploradas uma a uma, com suas características de implementação, motivos, análise de falhas, cada classe desenvolvida é inserida no contexto do sistema, relacionando-se com as demais. Ali é descrita a geração dos *tickets*, sua importância como identificadores únicos de cada transmissão, seu sistema de validação, o modo como são transformados pela serialização para serem anexados à mensagem, antes da assinatura digital e da cifragem.

[1] *http://www.fsf.org/copyleft/gpl.html*

Os métodos criptográficos de cifragem e assinatura digital também são tratados no Capítulo 3, onde são expostas suas relações com a biblioteca de criptografia utilizada, e são apresentados dados concretos quanto ao custo dos processos criptográficos, para melhor auxiliar a decisão sobre a relação segurança-eficiência representada pelo comprimento das chaves criptográficas.

Também é analisada a definição do suporte CORBA ao sistema desenvolvido, examinando os objetivos e necessidades para a comunicação através desse *middleware*, e como foram descritas e construídas as interfaces de comunicação para o sistema. É também apresentado o sistema de redirecionamento de conexões, implantado no servidor da aplicação, e que torna o sistema apto a operar através de um *firewall*.

O Capítulo 4 refere-se à modelagem da arquitetura projetada. A maneira como foram projetadas as classes e suas relações, as restrições impostas pelos sistemas utilizados, que modificam o sistema projetado, todos são apresentados de forma mais específica. Sendo parte essencial do sistema, uma grande atenção é dada à tarefa de adicionar criptografia e autenticação aos dados, e para isso são apresentadas as estruturas responsáveis por esses procedimentos, situando-as dentro do contexto do trabalho e refletindo as características dos modelos utilizados. Também é dado um grande enfoque sobre o contrato entre cliente e servidor para simular as funções de uma conexão *socket* tradicional, conferindo transparência à programação.

Os novos conceitos de estruturação das classes, exigidos pelo CORBA para a modelagem cliente-servidor, e a decisão de associar um serviço extra do CORBA, o *Factory Server*, são apresentados também no Capítulo 4.

No Capítulo 5 são apresentadas as aplicações estendidas através do uso das classes desenvolvidas. As duas aplicações, um emulador de terminal *TN3270* e um emulador de *telnet*, são apresentados como exemplo da facilidade e simplicidade de adaptação que o sistema de segurança desenvolvido propicia. As suas estruturas são apresentadas, como parte do estudo desenvolvido, situando as modificações e os possíveis avanços a serem realizados em trabalhos futuros.

Por fim, o Capítulo 5 discute sobre os *browsers* compatíveis com Java que foram testados e que, ao contrário da frase "*Write once, run anywhere*" (Escreva uma

vez, execute em qualquer lugar), divulgada em apoio ao Java pela *Sun Microsystems*, apresentam grandes incompatibilidades na execução dos programas Java.

2 REVISÃO TEÓRICA

2.1 Criptografia

O termo criptografia origina-se da união das palavras gregas *kryptós* (que significa escondido, oculto) e *graphos* (escrita) e designa a ciência de escrever em código, ou seja, tornar uma mensagem originalmente clara em um equivalente incompreensível, sendo que apenas o destinatário da mensagem consiga decifrar e compreender o seu significado [WEB 94]. Para decifrar a mensagem, normalmente utiliza-se uma chave, uma senha que permite reconstituir a mensagem recebida.

Outro conceito relacionado é a criptoanálise (também do grego *kryptós* e *analysis*, que significa decomposição). A criptoanálise identifica o conjunto de técnicas que servem para determinar a chave de uma mensagem, ou a própria mensagem, mesmo sem o conhecimento da chave.

A utilização da criptografia remonta a vários séculos, desde os babilônicos e os gregos. No entanto, sua capacidade de esconder o conteúdo, bem como a própria capacidade de analisar e quebrar a mensagem sempre esteve ligada à capacidade de cálculo manual do homem. Somente a partir da Primeira Guerra Mundial iniciou-se o emprego de máquinas mecânicas para acelerar esses processos.

Atualmente os métodos criptográficos tradicionais cederam lugar à criptografia computacional, onde as operações são implementadas por um computador ou por um circuito integrado especial. Isso acelera muito os processos de cifragem e decifragem (e também aumenta o poder da criptoanálise), e com isso diversos passos de encriptação (entre os mais comuns estão a substituição, a transposição e a composição matemática), de modo a cada vez mais mascarar os símbolos utilizados, e dificultar a sua análise [WEB 97].

Nessa tarefa de dificultar a análise é dado um enfoque especial à freqüência de caracteres em uma mensagem. Se uma criptografia mantém apenas substituições e transposições simples, ainda há uma possibilidade de descobrir a mensagem observando-se a freqüência da ocorrência dos caracteres na mensagem. Foi desse modo que em 1922 Herbert Yandley quebrou o código diplomático do governo

japonês. Sem ao menos conhecer a língua japonesa, Yandley usou uma tabela de freqüência de caracteres do idioma japonês [WEB 94]. Esse tipo de tabela já era conhecida e difundida. O código Morse, por exemplo, inventado ainda no século XIX, utilizava a tabela de freqüência da língua inglesa para economizar energia nas transmissões das mensagens. Um sistema ideal de criptografia deve, portanto, tentar também gerar símbolos com uma distribuição homogênea na mensagem.

Outro fator importante refere-se às chaves. Considerando que a robustez de um método criptográfico corresponda à quantidade de esforços (e portanto, tempo) necessários para quebrá-lo, bastaria definir uma chave com um comprimento extremamente grande. No entanto, graças ao avanço da capacidade de processamento dos computadores, é possível que haja a necessidade de redefinir as chaves a cada certo período de tempo. Isso leva a dois problemas:

- Existe uma grande dificuldade em "apenas" trocar as chaves periodicamente, quando o sistema de criptografia utiliza dispositivos em hardware, o que obrigaria a troca de dispositivos de todos usuários. Mesmo que a troca de chaves fosse possível, como nos sistemas implementados em software, ainda restaria a todos usuários a tarefa de obter essa nova chave.
- Oo segundo problema se refere ao gerenciamento da distribuição confiável das chaves [WEL 97]. Uma distribuição mal feita da chave possivelmente oferecerá mais riscos do que a própria possibilidade de quebra da chave.

Considerando essa dificuldade em distribuir novas chaves, o mais comum é definir uma "janela de tempo", na qual a criptografia ainda pode ser considerada segura. Se por exemplo, um dado deve permanecer seguro somente até ser processado e retornado, pode-se considerar aceitável manter uma chave que vai demorar dois anos para ser quebrada por um ataque. Com isso, reduz-se a necessidade da atualização e da distribuição de novas chaves.

2.1.1 Operações de Cifragem

Segundo [WEB 97] as operações usualmente efetuadas sobre os dados podem ser classificadas segundo as seguintes categorias:

- Cifras de Substituição: ocorre a troca de cada caracter ou grupo de caracteres por outro, de acordo com uma tabela de substituição. Pode-se quebrar este método analisando-se as freqüências dos caracteres no texto cifrado.
- Substituição Monoalfabética: cada letra do texto original é trocada por outra de acordo com uma tabela e com sua posição no texto. A Cifragem de César é um exemplo de substituição monoalfabética que consiste em trocar cada letra por outra que está 3 letras adiante na ordem alfabética, fazendo por exemplo A = D. Pode-se utilizar outro valor diferente de 3, o que constitui a chave da cifragem. Considerando apenas o alfabeto, existem apenas 26 chaves, o que torna esse método de baixa segurança.
- Substituição por Deslocamentos: a chave indica quantas posições deve-se avançar no alfabeto para cada caracter. Sendo uma chave igual a 020813, pode-se trocar a primeira letra pela que está duas posições à frente, a segunda pelo caracter 8 posições à frente e assim por diante, repetindo a chave se necessário.
- Substituição Monofônica: Como a anterior, mas agora cada caracter pode ser mapeado para um ou vários caracteres na mensagem cifrada (A = Z1, por exemplo). Isso evita a linearidade da substituição.
- Substituição Polialfabética: combinação no uso de várias substituições monoalfabéticas, usadas em rotação de acordo com um critério ou chave. Por exemplo, pode-se usar 4 tabelas, usadas alternadamente a cada 4 caracteres.
- Substituição por Polígramos: utiliza um grupo de caracteres ao invés de um caracter individual. Se fossem considerados trigramas, por exemplo, ABA poderia ser substituído por RTQ ou KXS.
- Cifras de Transposição: troca-se a posição dos caracteres na mensagem. Por exemplo, pode-se rescrever o texto percorrendo-o pelas colunas.
- Máquina de Cifragem: um código trabalha com grupos de caracteres de tamanho variável, ao contrário da cifra. Cada palavra é substituída por outra. Quebrar um código equivale a quebrar uma gigantesca substituição monoalfabética onde as unidades são palavras e não caracteres. Para isso, deve-se usar a gramática da língua e analisar a estrutura das frases. Tem o nome de Máquina de Cifragem devido às máquinas mecânicas usadas no início do século, que se baseavam em

engrenagens de tamanhos diferentes, girando a velocidades diferentes, e obtendo uma substituição polialfabética com chaves de $26n$, onde n é o número de engrenagens.

- Operações matemáticas: muitas operações matemáticas podem ser aplicadas à mensagem (usualmente a nível de bits ou bytes). Podem ir desde o OU-Exclusivo, troca da base numérica ou fatoração do produto de grandes números primos.

2.1.2 Criptoanálise

Um sistema de criptografia deve ser seguro mesmo quando os algoritmos de cifragem e de decifragem sejam conhecidos, e por essa razão são usadas chaves.

Uma pessoa não autorizada que tem acesso a alguns dos elementos de um criptosistema é denominada atacante. Um atacante passivo somente obtém cópias dos elementos, enquanto um atacante ativo pode alterar alguns desses elementos. Existem cinco tipos mais comuns de ataque (criptoanálise) [WEB 97]. Todos eles supões que o atacante possua conhecimento total sobre os métodos de cifragem e decifragem utilizados, mas não sobre as chaves:

- Ataque do texto cifrado: o criptoanalista tem à sua disposição uma grande quantidade de mensagens cifradas, mas desconhece as originais e as chaves utilizadas. Sua tarefa é recuperar as mensagens normais (deduzindo as chaves utilizadas).
- Ataque do texto conhecido: o criptoanalista tem à sua disposição uma grande quantidade de mensagens cifradas e também suas mensagens originais equivalentes. Seu objetivo é deduzir as chaves usadas (ou um método para recuperar mensagens cifradas com a mesma chave).
- Ataque adaptativo do texto escolhido: no método anterior, o criptoanalista poderia ser capaz de fornecer somente uma grande quantidade de mensagens de uma só vez. Agora, ele pode fornecer um pequeno conjunto, analisar os resultados, fornecer outro conjunto e assim por diante. Seu objetivo é deduzir as chaves utilizadas.

- Ataque do texto cifrado escolhido: o atacante não só tem uma grande quantidade de mensagens e seus equivalentes cifrados, mas pode produzir uma mensagem cifrada específica para ser decifrada e obter o resultado produzido. É utilizado quando se tem uma “caixa preta” que faz decifragem automática. Sua tarefa é deduzir as chaves utilizadas.
- Ataque da chave escolhida: o criptoanalista pode testar o sistema com diversas chaves diferentes, ou pode convencer diversos usuários legítimos do sistema a utilizarem determinadas chaves. Neste último caso, a finalidade imediata seria de decifrar as mensagens cifradas com essas chaves.

Nota-se que o ataque por força bruta não foi enumerado entre esses métodos, pois é o pior método possível. Considera-se que um sistema de criptografia foi quebrado quando se consegue determinar seu segredo por um meio mais inteligente que o ataque por força bruta.

2.1.3 Chaves Secretas e Públicas

A maior diferença entre os métodos de criptografia refere-se à existência de chaves públicas e secretas [WEL 97].

No sistema tradicional, também conhecido como criptografia de chaves simétricas, as chaves de cifragem e decifragem são idênticas e devem ser mantidas em segredo (chaves secretas).

No sistema que envolve chaves públicas, também chamadas de criptografia assimétrica, cada indivíduo tem um par de chaves: uma não secreta, usada para encriptar os dados, e uma chave privada, que é usada para a decriptação. A chave para a cifragem é distribuída livremente, recebendo a denominação de chave pública, e com ela qualquer pessoa pode encriptar uma mensagem para o proprietário das chaves, mas somente esse proprietário pode lê-la com sua chave privada.

2.1.4 Algoritmos de Chave Única ou Secreta (Simétricos)

O exemplo mais difundido de sistema criptográfico de chave única é o DES (*Data Encryption Standard*), desenvolvido pela IBM e adotado como padrão nos Estados Unidos em 1977. O DES cifra blocos de 64 bits (8 caracteres) usando uma chave de 56 bits mais 8 bits de paridade, perfazendo um total de 64 bits. O algoritmo inicia realizando uma transposição sobre os 64 bits da mensagem, seguido de 16 passos de cifragem e conclui realizando uma transposição que é a inversa da transposição inicial [WEB 97].

Para os 16 passos da cifragem usam-se 16 sub-chaves, todas derivadas da chave original através de deslocamentos e transposições.

Um passo da cifragem do DES tem dois objetivos básicos: a difusão e a confusão. A difusão visa eliminar a redundância existente na mensagem original, distribuindo-a pela mensagem cifrada. O propósito da confusão é tornar a relação entre a mensagem e a chave tão complexa quanto possível.

O DES pode ser quebrado pelo método da força bruta, tentando-se todas as combinações possíveis para a chave. Como a chave tem 56 bits, têm-se um total de 2^{56} chaves possíveis.

Utilizando-se um ataque de texto plano escolhido e adaptativo, onde o atacante pode fornecer um texto para ser cifrado e obter a saída correspondente, e além disso realizar essa operação quantas vezes quiser, adaptando a entrada para testar diversas hipóteses, já foi demonstrado que o DES pode ser quebrado com 2^{47} ou até mesmo 2^{43} tentativas.

Recentemente, um concurso promovido pela RSA Data Security, chamado *DES Challenge II*[2] promoveu a quebra do DES em menos de três dias (56 horas). Não foi considerado um ataque por força bruta, pois utilizaram-se algumas características do DES para determinar os melhores pontos de ataque. Para isso, o grupo da Eletronic Final Frontier[3] desenvolveu uma máquina com arquitetura capaz de analisar todas as possibilidades do DES em cerca de 9 dias. E o custo total dessa máquina é de cerca de 50 mil dólares [MOR 98].

---

[2] *http://www.rsa.com/DES*
[3] *http://www.eff.org*

Diversos algoritmos de cifragem de blocos com chave única foram ou estão sendo propostos. A Tabela 1 mostra algumas características dos algoritmos mais encontrados na bibliografia[WEB 97].

Tabela 1 _ Alguns Sistemas de Chave Única

IDEA	Blocos de 64 bits com chave de 128 bits
Triple-DES	O DES é aplicado 3 vezes. Precisaria 2^{112} tentativas para quebrar.
Lucifer	Base do DES, tem 128 bits, mas é considerado mais fraco que o DES. A quebra precisaria de 2^{33} tentativas
Madryga	Blocos de 8 bits sem transposição, somente ou-exclusivo e deslocamento de bits. Fraco.
NewDES	Baseado no DES, usa blocos de 64 bits e chaves de 120 bits. Opera a nível de bytes. Um pouco mais fraco que o DES.
FEAL-N	Projetado para se mais simples e rápido que o DES. Possui um número variável de passos. Blocos e chaves de 64 bits, mas é muito fraco com menos de 8 passos.
REDOC II	Operam somente com bytes. Usa blocos de 80 bits sobre uma chave de 160 bits, com substituições, transposições e ou-exclusivo. Considerado bem seguro.
LOKI	Semelhante ao DES, trabalha com blocos e chaves de 64 bits.
Khufu	Patenteado pela IBM, pretende eliminar alguns pontos fracos do DES. Tabelas de substituições de 256 posições de 32 bits (o DES tem 6 posições com 4 bits). Chaves de 512 bits e número de passos flexível (múltiplos de 8).
MMB	Semelhante ao IDEA, usa blocos de 128 bits, chaves de 128 bits e sub-chaves de 32 bits.
Skipjack	Desenvolvido pelo NSA, usa chaves de 80 bits e realiza 32 passos de cifragem. Ainda mantido como segredo. Possivelmente será implementado em hardware.

2.1.5 Algoritmos de Chave Pública (Assimétricos)

Na criptografia tradicional, com uma chave única, há sempre o risco envolvido na divulgação da chave entre o remetente e o destinatário da mensagem. As exigências de controle sobre a transmissão dessas chaves, bem como sobre a armazenagem delas (principalmente em sistemas computacionais acessados por vários usuários), muitas vezes tornam seu uso muito mais complexo.

Para solucionar esse problema de gerenciamento, Whitfield Diffie e Martin Hellman e independentemente Ralph Merkle introduziram o conceito de chaves públicas, em 1976. Sistemas criptográficos de chave pública têm dois usos primários, a criptografia e a autenticação. Por meio desse sistema, cada usuário tem um par de chaves, uma chamada chave pública e a outra privada. A chave pública é distribuída, e a privada guardada em segredo. A necessidade do emissor e do receptor da mensagem compartilharem dados sobre a chave de encriptação é eliminada, pois toda comunicação envolve apenas as chaves públicas, e as chaves privadas jamais trafegam. Assim, acaba a necessidade de garantir um meio de transmissão confiável. A única ressalva fica quanto à identificação do proprietário das chaves, que pode ser resolvido depositando-se e retirando as chaves de um site confiável. Qualquer pessoa pode cifrar uma mensagem com a chave pública, mas somente o proprietário da chave privada poderá decifrá-la, e ler seu conteúdo [WEB 97].

Num sistema criptográfico de chaves públicas sempre há uma relação matemática entre a chave pública e a privada, e dessa forma seria possível descobrir a chave privada baseando-se na chave pública distribuída. A técnica utilizada para dificultar essa ação é tornar a derivação da chave privada a partir da chave pública o mais difícil possível. Em alguns sistemas costuma-se utilizar a fatoração modular de um produto de números primos. As chaves pública e privada são funções de um par de números primos grandes (entre 100 a 200 dígitos decimais). Para gerar as duas chaves, escolhe-se inicialmente dois números primos, p e q. A seguir calcula-se $n = p \times q$. Então escolhe-se randomicamente um número e, tal que e e $(p-1)(q-1)$ sejam primos entre si, ou seja, não tenham fatores comuns. Depois, calcula-se um número d tal que $(e \times d)$ módulo $(p-1)(q-1)$ seja igual a 1, ou seja, $e \times d = 1 \bmod (p-1)(q-1)$. Os números e e n formam a chave pública; os números d e n são a chave privada. Os números p e q não são mais necessários e devem ser esquecidos.

Para cifrar uma mensagem M, primeiro divide-se esta mensagem em blocos de t bits, onde t é calculado de tal forma que o valor numérico de cada bloco (o valor obtido interpretando-se a sua cadeia de bits como um número) seja menor que n. Em termos práticos, utiliza-se o maior valor para t de tal forma que a condição $2^t < n$ ainda seja satisfeita. Sob o ponto de vista decimal, se n tiver 200 dígitos, então cada bloco m_i também terá cerca de 200 dígitos, mas sempre de forma que $m_i < n$. A mensagem cifrada C também será formada por blocos c_i, cada um calculado por $c_i = m_i^e \bmod n$, ou seja eleva-se o bloco numérico m_i a potência e, em aritmética módulo n.

Para decifrar a mensagem C, basta tomar cada bloco numérico cifrado c_i e calcular $m_i = c_i^d \bmod n$, ou seja, eleva-se c_i a potência d, em aritmética módulo n. Já que nesta aritmética módulo n, devido à escolha de d e e, tem-se que $c^d = (m^e)^d = m^{ed} = m^{k(p-1)(q-1)+1} = mx1 = m,$ a fórmula recupera a mensagem. Note-se que a mensagem poderia, da mesma forma, ser cifrada com d e decifrada com e, a escolha é arbitrária.

A segurança do RSA está baseada no problema de fatorar números grandes. No presente momento, o melhor algoritmo de fatoração tem um tempo de solução na ordem de $O(e^{\text{srqt}(\ln n \text{ x } \ln(\ln n))})$. Se n tem 200 dígitos (aproximadamente 664 bits), a fatoração levará na ordem de $2{,}7x10^{23}$ passos. Assumindo-se um computador que possa realizar um milhão de passos por segundo (uma suposição generosa, dada a magnitude dos números envolvidos), levará $3{,}8x10^{9}$ anos para a fatoração. Mesmo que o poder computacional aumente um milhão de vezes, a fatoração ainda consumiria 4 milhões de anos. E se mais segurança for necessária, basta aumentar n por alguns bits [WEB 97].

Sendo de custo extremamente reduzida a multiplicação de dois números, não há problemas em gerar a chave, mas a fatoração da chave pública (para descobrir os dois números primos) é uma operação muito demorada. Na decodificação da mensagem, basta saber a chave privada, que a operação resume-se a uma divisão matemática, também de custo muito baixo. Essa é a base do sistema RSA [RSA 98].

Outro tipo de relação matemática muito usada é a dos logaritmos discretos. Enquanto no sistema anterior usava-se uma operação de multiplicação entre dois números primos, nesse os números são envolvidos em operações de exponenciação. A

quebra desse número gerado se dá através da operação do logaritmo, também muito demorada. Os sistemas ElGamal e DSA utilizam essa técnica.

Os maiores índices de fatoração até hoje foram de chaves com 426 bits para a multiplicação de números primos e a de 503 bits para o método do logaritmo discreto. E a quebra da chave de 426 bits levou 8 meses, numa cooperação de 1600 computadores via Internet [RSA 98].

Recentemente, pesquisadores estão desenvolvendo sobre a teoria matemática das curvas elípticas, e a maior chave quebrada até agora continha 79 bits [RSA 98].

Um fator de desvantagem para os algoritmos assimétricos refere-se à velocidade de cifragem. Atualmente, um algoritmo RSA implementado em software é cerca de 100 vezes mais lento que a implementação do DES, produzindo uma taxa de cifragem de 7,4 kbits/s para uma chave de 1024 bits. Implementado em hardware, o RSA é entre 1000 e 10000 vezes mais lento que o DES, com uma taxa de 300 kbits/s para uma chave de 512 bits [RSA 98].

Tabela 2 _ Alguns Sistemas de Chave Pública

RSA	Método mais popular. Utiliza o sistema de fatoração modular de números primos.
ElGamal	Utiliza o método do logaritmo discreto.
Merlke-Hellman	Utiliza um subgrupo da soma combinatorial para resolver através do Problema da Mochila
Chor-Rivest	Outro sistema que utiliza o Problema da Mochila. Utiliza produto não modular.
LUC	Utiliza operações de recorrência sobre uma Sequência de Lucas[1]
McEliece	Baseado na Teoria de Codificação Algébrica

[1] $T_n = PT_{n-1} - QT_{n-2}$, onde P e Q são número primos

O sistema de criptografia escolhido para a implementação desse trabalho é o RSA. Sua escolha foi condicionada por diversos fatores, sendo o principal a

necessidade de trafegar as chaves, quando do início da conexão, sem ter que criar sistemas complexos de garantia e gerência do transporte das chaves. Quando utiliza-se o RSA, basta enviar as chaves públicas para o receptor. Outro fator importante é a capacidade de alterar o tamanho da chave, em tempo de compilação, de acordo com as exigências de segurança e velocidade. Além das características inerentes a um sistema de chave pública, o RSA foi escolhido, entre os outros sistemas com essas características, pela popularidade e por ter implementações disponíveis em diversas linguagens. Assim, pode-se num trabalho futuro rescrever partes do sistema, em especial do servidor, portando-o de Java para C++, que é mais eficiente, por não ser interpretado.

A implementação do RSA escolhida para o trabalho foi o pacote Cryptonite, de Logi Raginarsson[4]. Esse conjunto de classes Java para criptografia é disponibilizado segundo a licença GPL - *General Public License* da *Free Software Foundation*[5], que garante a disponibilidade do código fonte e uso livre.

2.2 Autenticação

As ferramentas de autenticação possibilitam determinar a identidade das partes em uma negociação e assegurar que as mensagens são originárias de quem se espera. Assim, a autenticação é usada como base para a autorização (determinações de quais privilégios devem ser garantidos para um usuário ou processo em particular), privacidade (evitando que informações sejam conhecidas por não participantes de um grupo) e não repudiação (depois de assinada, a mensagem não pode ser negada) [WEL 96].

Todos esquemas de autenticação são baseados na posse de alguma informação secreta conhecida somente pelo usuário e possivelmente (mas não necessariamente) pelo próprio sistema de autenticação. Isso significa que o segredo não pode ser compartilhado, devido à possibilidade de alguém vir a personificar o usuário em uma

---

[4] *http://www.hi.is/~logir/*
[5] *http://www.fsf.org/copyleft/gpl.html*

negociação futura. Assim, o que diferencia os diversos sistemas de autenticação reside em como provar que se sabe o segredo sem mostrá-lo.

Sistemas de autenticação também provêm diferentes níveis de funcionalidade. No mínimo, permitem ao destinatário verificar a origem da mensagem de um usuário determinado. Sistemas mais poderosos asseguram que as mensagens não poderão ser copiadas e reenviadas no futuro, adicionando um *timestamp* da criação da mensagem. A funcionalidade de não repudiação permite a um servidor provar que o cliente enviou a mensagem, e que essa não foi forjada. Assim, pode-se determinar com certeza quem requisitou um serviço, e sua implicação no sistema. Os sistemas de autenticação podem ainda permitir que múltiplos usuários validem essa mensagem, e que devam estar presentes no mínimo *n* usuários para extraí-la [WEB 97].

2.2.1 Conceitos

Existem três principais formas de autenticação: senhas, *tickets* de sessão e assinatura digital.

No sistema de senhas, freqüentemente a senha é armazenada no servidor (usualmente de forma criptografada). A transmissão da senha para o servidor, que a compara com a senha já armazenada, identifica o usuário. No entanto, esse sistema é muito suscetível à interceptação da transmissão, tornando-se muito inseguro e inaceitáveis em um ambiente de rede que requer privacidade [WEL 96].

No sistema de *tickets* de sessão, que entre as muitas implementações destaca-se o sistema de Needham e Schroeder, usado no *Kerberos*, a informação secreta usada para a verificação nunca é transmitida abertamente, e nunca é vista pelo destinatário. Para isso, um terceiro elemento, chamado "Servidor de Autenticação" cria diversos *tickets* de sessão (*ticket* esse criado a partir das informações secretas do remetente e do destinatário, conhecidas pelo servidor), e que são usadas pelas partes para a autenticação da mensagem durante a negociação. As informações da sessão só são úteis para os participantes, e os *tickets* podem ser adicionados de um *timestamp* para proteger contra reenvio. Novas negociações (inclusive entre os mesmos clientes e servidores) requerem novas senhas [WEL 96].

Finalmente, a assinatura digital, que pode ser realizada de duas formas. Através de sistemas de criptografia de chave pública, ou do uso de funções unidirecionais.

Os sistemas que usam o conceito de chave pública, devido à relação matemática entre as chaves pública e privada, normalmente permite além do fluxo D(E(m)) = m o fluxo E(D(m)) = m, ou seja, no segundo método primeiro cifra-se a mensagem com a chave privada, e quem a decifrar com a chave pública vai ter a garantia da origem da mensagem, pois a chave privada usada para cifrar a mensagem deve ser mantida em segredo pelo proprietário. Esse sistema, presente no RSA e em alguns dos mais comuns sistemas de criptografia de chave pública torna os processos de criptografia e autenticação muito similares, facilitando seu uso [WEL 96].

Esse método, no entanto, pode apresentar algumas desvantagens. Em implementações práticas, o uso de algoritmos de chave pública para assinatura digital são muitas vezes ineficientes para assinar documentos longos. A fim de reduzir o tempo necessário, muitos protocolos de assinatura digital são implementados com funções de *hash* unidirecionais [WEB 97]. No lugar de assinar um documento, calcula-se o *hash* e aplica-se a assinatura somente sobre o *hash*, que normalmente é de tamanho reduzido (128 a 512 bits). O processo segue então os seguintes passos:

• o usuário A produz um *hash* unidirecional do documento.

• o usuário A assina o *hash* com sua chave privada, assinando assim o documento.

• o usuário A envia o documento e o *hash* assinado para o usuário B.

• o usuário B calcula o *hash* do documento. A seguir, B restaura o *hash* assinado, usando a chave pública de A. Se o *hash* produzido e o *hash* restaurado foram iguais, então o documento e a assinatura são válidos.

Este protocolo tem outras vantagens adicionais. O documento pode ser armazenado em forma legível, o que facilita o acesso e a leitura. Documento e *hash* podem ser guardados em locais diferentes, o que dificulta mais ainda adulterações. O espaço necessário para guardar o *hash* é bem reduzido. E quanto à privacidade e outros aspectos legais o documento pode ser mantido secreto; somente o seu *hash* assinado necessita ser tornado público. Só quando a autoria de um documento (ou de uma idéia) deve ser provada é que o documento precisa ser tornado público.

Alguns dos sistemas de *hash* mais usados são apresentados na Tabela 3 [WEB 97] e [RSA 98].

Tabela 3 _ Alguns Sistemas de *Hash*

MD4 (Message Digest 4)	Criado por Ron Rivest (do RSA). Produz um *hash* de 128 bits. Trabalha sobre blocos de 512 bits, divididos em 16 sub-blocos de 32 bits. Utiliza operações de soma, deslocamento, ou-exclusivo e operações lógicas “e” e “ou”.
MD5 – Mais utilizado	Melhoramento do MD4, inclui mais funções de embaralhamento e mais rodadas.
SHA	Projetado pelo NIST e pelo NSA, é inspirado no MD5. Apresenta no entanto um *hash* de 160 bits.
Snefru	Produz um *hash* de 128 ou 256 bits. A mensagem é dividida em blocos de 384 ou 256 bits, respectivamente. A segurança é baseada em uma função que randomiza os dados em vários passos. Cada passo tem 64 etapas, e cada etapa é constituída de uma cifragem por substituição combinada com ou-exclusivo e rotação. Mínimo 8 passos.
N-HASH	Desenvolvidos pelos mesmos autores do N-FEAL. Trabalha sobre blocos de 128 bits, e seu *hash* também tem 128 bits. Utiliza funções de soma, ou-exclusivo e rotação. Mínimo 15 passos.
HAVAL	Apresenta *hash* de comprimento variável. Usa blocos de 1024 bits, e com um número variável de passos (entre 3 e 5) pode produzir *hash* de 92, 128, 160, 224 ou 256 bits. Baseado no MD5, é mais rápido.

Este trabalho implementa a autenticação por chaves públicas, devido à necessidade de transmitir a mensagem também (num *hash*, teria que transmitir a mensagem e o *hash*). Essa escolha tabém é condicionada pelo tipo de mensagem trafegada normalmente, que no máximo compreende poucas centenas de bytes, e não traz grandes prejuízos ao desempenho.

Além disso, também são adicionados *tickets* de identificação, que contém um *timestamp*, e que são gerados a cada transmissão de dados. Cada parte tem seu *ticket*, e renova-o cada vez que vai transmitir uma nova mensagem. Com isso, garante-se a unicidade da mensagem transmitida, evitando a retransmissão de mensagens.

A mensagem e os *ticket* são assinados com a chave privada do remetente e então cifradas com a chave pública do destinatário, para a transmissão, **Ed(Dr(m,Td,Tr))**.

O destinatário recebe essa mensagem, e fazendo o processo contrário Dd(Er(**Ed(Dr(m,Td,Tr))**)) obtém a mensagem, e os dois *tickets*.

2.3 A Linguagem Java

A linguagem Java foi desenvolvida pela *Sun Microsystems* e lançada oficialmente em 1995. O Java é uma linguagem orientada a objetos, com uma sintaxe muito próxima à da linguagem C. Ela logo conquistou muitos adeptos e tornou-se uma referência para as aplicações voltadas à Internet [PIN 97].

Segundo [PIN 97], a linguagem Java é uma linguagem multiplataforma, onde um mesmo código compilado pode ser executado em diversas plataformas, indo desde PCs com sistema *Windows95* ou NT, a estações de trabalho com sistema *Solaris*. Essa multiplataforma foi obtida graças à combinação entre compilação e interpretação presente na linguagem. Um programa fonte em Java passa inicialmente pelo compilador, onde é gerado um arquivo em um formato intermediário conhecido como *bytecodes,* que depois é interpretado. Os *bytecodes* nada mais são do que código executável para a Máquina Virtual Java _ *Java Virtual Machine* (JVM), que é implementada pelo interpretador Java na máquina hospedeira.

A linguagem Java incluiu em sua definição todos os principais conceitos da orientação a objetos e criou o conceito dos *applets*. Um *applet* é um pequeno programa capaz de ser inserido dentro de uma página HTML e transferido pela rede junto aos demais dados desta, e executado pelo próprio *browser,* que deve implementar a JVM, na própria máquina cliente. O conceito de *applets* junto com a multiplataforma garantiram ao Java enorme disseminação entre os desenvolvedores de páginas e aplicativos para a Internet.

Outro ponto forte da linguagem Java é o quesito segurança. Um *applet*, ao executar em uma máquina cliente, não tem livre acesso a esta. Sua ação sofre diversas restrições de segurança que a impedem de realizar determinados atos considerados ofensivos ou perigosos. A princípio o *applet* pode somente consultar ou abrir conexões com sua máquina origem (servidora), e não pode abrir ou gravar arquivos na máquina cliente. Essas restrições de segurança dificultam seriamente a criação de programas nocivos, ou até mesmo vírus, em Java, que possam ser distribuídos através de *applets*. Essa característica deu tranqüilidade a muitos usuários na Internet e ajudou

ainda mais a popularização desta linguagem, mas restringiu as possibilidades de desenvolvimento de aplicações mais complexas.

Sendo o objetivo deste trabalho estender as funcionalidades de comunicação por *sockets,* através do acréscimo de criptografia e autenticação, deparou-se com um problema. Sendo os *browsers* os principais ambientes de execução de *applets* Java, a comunicação deverá estar sujeita às restrições de segurança já citadas, o que de certa forma tornaria o uso das aplicações desenvolvidas muito mais inflexível, pois cada máquina servidora deveria ter seu servidor *http* para que o *applet* pudesse se comunicar com essa máquina. Soma-se a isso a atual tendência de proteger a rede interna das empresas com o uso de *firewalls*, e o que resta são duas escolhas aparentemente excludentes: utilizar *applets* para distribuir as aplicações, e proteger a rede interna.

A melhor solução encontrada para esse problema é a de uma estrutura de redirecionamento, que denomina-se *tunelamento* ou *servidor proxy* [VOG 97a], onde um processo fica em uma máquina visível na rede, especificamente o servidor do *applet* Java, e redireciona as conexões de rede para seu destino. Para implementar esse *middleware*, muitas opções são possíveis [ORF 97], desde programas que lidam diretamente com as conexões *socket*, até estruturas mais avançadas, notadamente CORBA e RMI. A escolha do uso do CORBA para a implementação desse trabalho baseou-se principalmente pela flexibilidade que o CORBA oferece, por ser independente de linguagem, de plataforma, e por permitir uma adição de características tais como suporte a transações, ligação dinâmica, e outros. Essa flexibilidade garante a possibilidade de desenvolver novos projetos baseados nas classes aqui apresentadas, adicionando características mais específicas que certas aplicações requerem. A estrutura da comunicação entre o *browser* e o serviço requisitado pode ser visualizada na Figura 1.

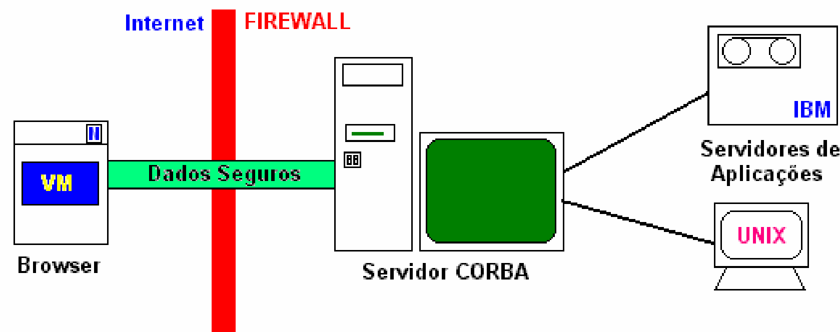


Figura 1 _ Estrutura de Comunicação

2.4 Sockets

Segundo [ORF 97], *sockets* são um sistema de comunicação ponto-a-ponto muito utilizados para programação em rede, pois abstrai os detalhes da rede, durante a programação.

Os *sockets* foram apresentados em 1981 como parte do sistema operacional Unix BSD 4.2, para prover uma interface genérica para o *Unix-to-Unix Interprocess Communications* (IPC). Por causa disso os *sockets* também são conhecidos como *Berkeley Sockets*.

Em 1985, a *Sun Microsystems* introduziu seus protocolos NFS e RPC utilizando *sockets*, e hoje os *sockets* são suportados por virtualmente todos sistemas operacionais, e também com suporte a outros protocolos de rede além do TCP/IP.

Java incluiu o suporte aos *sockets* como parte de suas classes *core*, e, hoje, pode-se afirmar que os *sockets* são o padrão de fato para aplicações portáveis sobre redes TCP/IP [ORF 97].

Os *sockets* dividem-se em três tipos de implementação: por conexão, por datagrama e de controle. Esses formatos têm grande relação com o TCP/IP, pois no modo conexão é utilizado o protocolo TCP para a comunicação (o que significa garantia de entrega, conexão *virtual* entre as partes), no modo datagrama é utilizado o protocolo UDP (sem garantia de entrega, mas muito mais rápido e suficientemente confiável em uma rede local), e no modo controle trafegam informações usando o protocolo ICMP.

Sendo um sistema portável e padronizado, é comum serem os *sockets* a interface de comunicação para aplicações Java que desejam se conectar a serviços de rede tais como *telnet, ftp, mail,* etc. Dessa forma, a maneira mais genérica de garantir serviços extras de segurança e autenticação a uma aplicação de rede é estender os *sockets*, pois além de alcançar o maior número de aplicações possíveis, ainda reduziria consideravelmente as modificações nas aplicações já existentes.

Por esse motivo, esse trabalho objetiva estender o padrão de *sockets* do Java, adicionando as características de segurança já mencionadas, mas sem perder a compatibilidade com o *sockets* padrão.

2.5 CORBA

O *Common Object Request Broker Architecture* (CORBA) é um dos mais importantes projetos de *middleware* da atualidade. Ele é o produto de um consórcio chamado *Object Management Group (OMG)*, que reúne mais de 700 companhias de informática.

O que faz o CORBA tão importante é sua definição de *middleware*, que tem potencial de substituir quaisquer outras formas de *middleware* cliente-servidor existentes [ORF 97]. Para isso, foi projetada uma arquitetura de interconexão conhecida como *Object Management Architecture* (OMA), conforme pode ser visualizado na Figura 2. Ou seja, CORBA usa objetos como metáforas de unificação entre aplicações e a rede. Um sistema CORBA é inteiramente auto-descrito, e a especificação do serviço é independente da implementação.

Essa especificação é descrita em uma linguagem neutra chamada *Interface Definition Language* (IDL) que define os métodos com os quais os objetos irão se comunicar. A implementação segue-se estruturando esses métodos. Assim, conversando através de interfaces bem definidas, as implementações podem ser desenvolvidas em linguagens, ferramentas, sistemas e redes totalmente independentes, com a garantia da comunicação. Através da especificação em IDL, pode-se mapear os objetos CORBA para diversas linguagens (entre as quais C++, Java, Perl, ADA, Smalltalk, COBOL, etc.), e garantir a compatibilidade de suas interfaces.

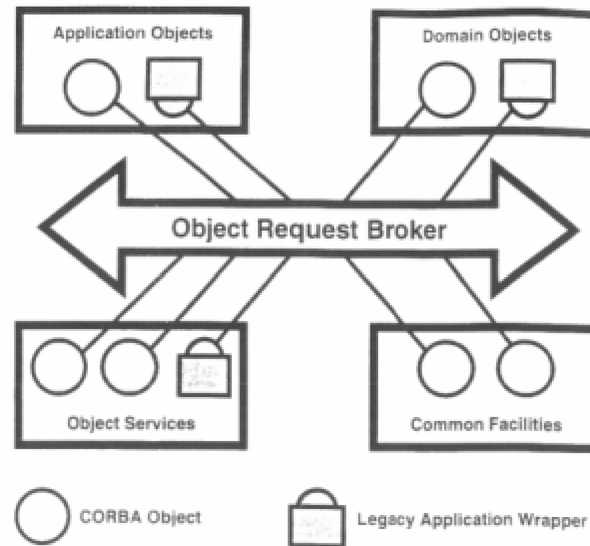


Figura 2 _ Arquitetura de Referência OMA [VOG 97a]

Um *Object Request Broker* (ORB) é um "bibliotecário" de objetos. Ele permite que os objetos da aplicação façam (e recebam) requisições transparentemente para outros objetos locais ou remotos. O cliente não necessita preocupar-se com os mecanismos de comunicação, de ativação ou de armazenamento nos objetos servidores, pois as interfaces CORBA realizam essas tarefas, utilizando a estrutura OMA estabelecida (Figura 3).

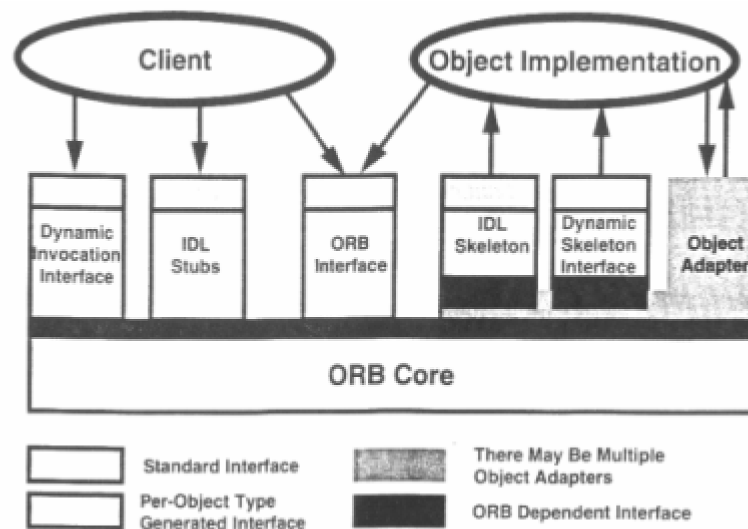


Figura 3 _ Interfaces CORBA [VOG 97a]

A especificação CORBA 1.1, de 1991, somente especificava a linguagem IDL, os mapeamentos para as linguagens e uma API para a comunicação com os ORBs. Então, pode-se escrever programas portáveis que podem rodar sobre diversos ORBs

compatíveis com o CORBA disponíveis no mercado [ORF 97]. Com a adoção da especificação CORBA 2.0, em 1994, garantiu-se também a comunicação entre objetos ligados a *object brokers* desenvolvidos por companhias diferentes.

Objetos CORBA são estruturas que podem existir em diferentes locais da rede. Eles são encapsulados como componentes binários que clientes remotos podem acessar via invocação de métodos. Dessa forma, a linguagem e o compilador usados para criar os objetos servidores são totalmente transparentes para os clientes.

Além das características de compatibilidade e portabilidade do CORBA, a OMG definiu ainda diversas funcionalidades conhecidas como *CORBAservices*. Esses serviços são coleções de funções a nível de sistema reunidas através de interfaces definidas em IDL. Muitos desses serviços são bastante conhecidos, como *Transaction Service*, o *Naming Service* (semelhante a um *Domain Name Service* de uma rede Internet), *Query Service*, *Persistence Service* e muitos outros. Especificamente, esse trabalho utilizará o serviço conhecido como *Factory Service*. Esse método permite que ao invés de termos apenas um servidor CORBA para o serviço, o que levaria à restrição de um cliente por vez no sistema (devido à característica de manter a conexão do *socket*), poderão existir diversos servidores, um para cada cliente, instanciados e inicializados de acordo com as requisições de conexão. O *Factory Service* encarrega-se de criar os servidores, e eliminá-los quando não são mais necessários aos clientes.

Neste trabalho foi utilizado o ORB denominado *ORBacus*, produzido pela *Object Oriented Concepts Inc.*[6]. Em sua versão 3.1.1, o ORBacus possui versões para C++ e Java com versões para diversos sistemas operacionais (entre eles o Windows, Linux, AIX, Solaris, IRIX, HP-UX e diversos outros Unix), e teve como grande atrativo a licença de uso e distribuição livre para fins não comerciais, a indicação de diversos sites sobre CORBA e o grande suporte aos usuários realizado diretamente pelos criadores do sistema, através de uma lista de e-mail.

---

[6] *http://www.ooc.com*

3 ANÁLISE DO SISTEMA

3.1 Análise Geral

A arquitetura proposta por esse trabalho visa a implementação de classes de comunicação em Java, com funções semelhantes aos *sockets TCP* da biblioteca padrão do Java, acrescidos de alguns recursos, tais como a autenticação, a criptografia e a assinatura digital, de forma a prover sigilo e segurança aos dados transportados pela rede, mantendo ainda a compatibilidade com o modelo de comunicação de rede mais utilizado, os *sockets*.

Essa compatibilidade possibilita, além da menor necessidade de aprendizado na hora de utilizar a arquitetura proposta, adaptar aplicações atualmente desenvolvidas com *sockets* para um nível mais sigiloso de comunicação dos dados sem a necessidade de mudanças estruturais significativas. A eficácia dessa adaptação é comprovada através de duas aplicações tradicionais de rede, os emuladores de terminal *telnet* e *TN3270*, que foram modificados com o mínimo de alterações e estão operando satisfatoriamente com as novas classes. O uso das classes de comunicação segura nessas aplicações será mais detalhado no Capítulo 5.

A decisão de adicionar características de criptografia e autenticação à comunicação já seria uma tarefa interessante. Afinal, dos níveis de comunicação, o mais apropriado para implementar a criptografia e a autenticação é o nível de transporte, onde se destaca a utilização de *sockets*. Nada mais óbvio, portanto, que a escolha de uma estrutura que simule os *sockets*, pois esses são amplamente utilizados na manutenção de conexões TCP, e possuem uma bem definida (e conhecida) interface de utilização. O *SSL* (*Secure Socket Layer*), sistema desenvolvido pela Netscape, também realiza as funções a nível de *sockets*, mas seu uso em *browsers* apresenta restrições ao comprimento das chaves utilizadas, impostas pelo governo norte-americano.

Para justificar sua realização, o sistema deve apresentar características inovadoras e vantajosas com relação aos sistemas já existentes. O *Kerberos*, por exemplo, utiliza um servidor de autenticação como terceiro elemento da autenticação de uma comunicação, e usa o DES como modelo de criptografia. Quando o servidor de autenticação recebe a requisição de um cliente, verifica sua senha e gera uma chave de sessão, encriptando-a com a senha do cliente. Além disso, uma cópia da chave de

sessão é encriptada com a chave secreta do serviço. Essas duas chaves são retornadas ao cliente. Este decripta a sua parte da mensagem, para verificar a autenticidade do servidor de autenticação, e então requisita a conexão ao serviço enviando a parte ainda encriptada da mensagem. Ao receber essa mensagem, o serviço deduz que o *ticket* é válido, pois somente o servidor de autenticação conhece sua chave secreta. A comunicação a partir daí utiliza a verificação da chave de sessão para autenticar a conexão. Essa autenticação pela troca de *tickets* cifrados é conhecida como Protocolo de *Needham e Schroeder*.

O uso do sistema do *Kerberos* acarreta a necessidade de ter previamente cadastrada a chave privada (ou um segredo equivalente) junto ao servidor de autenticação, com todos os riscos da troca de chaves secretas. Além disso, a transmissão da senha do cliente ainda é feita em aberto, gerando a necessidade de utilizar modelos mais complexos, como as senhas de uso único. Para utilizar as facilidades do sistema, tanto o serviço como o cliente devem ser adaptados para funcionar utilizando o protocolo do *Kerberos*.

Outro sistema muito utilizado é o *ssh* (*Secure Shell*). A criptografia utilizada na comunicação também e o DES, mas para garantir a troca das chaves secretas de forma segura é utilizado o RSA no início da negociação. O objetivo do *ssh* é prover uma forma segura de substituir os serviços de *rsh, rlogin e rcp,* e também é utilizado para cifrar os acessos remotos de terminais X-Windows. Não há a necessidade de um servidor de autenticação, mas é necessário um agente rodando tanto no lado do cliente quanto do servidor para armazenar e autenticar as chaves RSA. Uma aplicação *ssh* é específica para um tipo de serviço, e há a necessidade de clientes escritos proprietariamente para o *ssh*.

Frente a esses sistemas, a primeira característica marcante da arquitetura proposta é o uso de Java. No mundo interligado pela Internet em que se vive hoje, é considerável a habilidade de dispor de um serviço desejado em qualquer lugar do mundo. Infelizmente, não basta apenas adicionar características de criptografia e autenticação a um sistema baseado em *sockets* no Java. As mesmas restrições de segurança que tornam esta linguagem menos vulnerável, a tornam muito inflexível quanto às conexões de rede. Um *browser* que implementa as restrições de segurança sobre o Java (e pode-se afirmar que a maioria dos *browsers* disponíveis no mercado realizam essas restrições) só permite que o *applet* realize conexões de rede com o

computador de onde se originou a aplicação, como mostra a Figura 4. A utilização de sistemas implementados dessa forma torna-se muito inflexível, e exige a adição de serviços extras (um servidor *httpd*) para cada máquina provedora de um serviço específico, o que dificilmente constitui um atrativo para o uso de uma aplicação que use essa arquitetura.

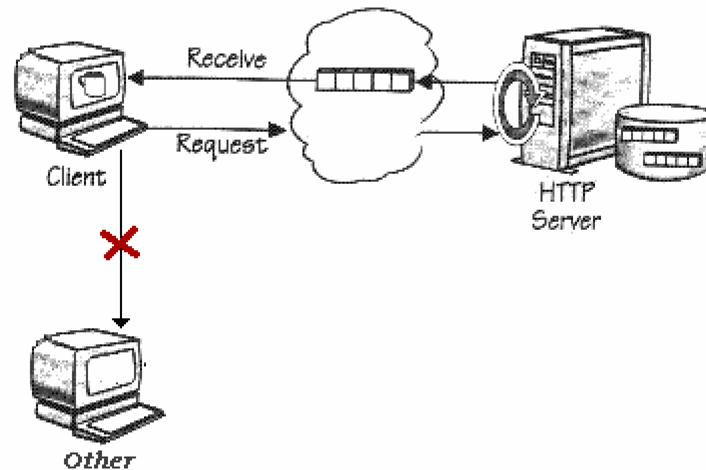


Figura 4 _ Conexões de um Applet Java

A solução adotada para esse problema é o uso de um sistema de redirecionamento de conexões. Esse sistema, especialmente localizado no servidor de origem do *applet* Java, recebe as conexões *socket* do *browser* e as direciona ao serviço final, fazendo o papel de uma ponte, um *gateway* [VOG 97a]. [BIR 95] define esse tipo de arquitetura como um *Wrapper Server* e a ela podem ser adicionadas diversas funcionalidades, tais como tolerância a falhas, criptografia e autenticação, serviços de *firewall* e muitos outros, de acordo com as necessidades dos sistemas envolvidos.

Devido à existência de um terceiro elemento na conexão, o servidor de redirecionamento, convencionou-se para esse trabalho considerar como **serviço** uma aplicação que provê funcionalidades específicas a quem se conecta a seu endereço (por exemplo, o *telnet* é um serviço de *login* remoto, para quem se conecta ao *host* hospedeiro do serviço, na porta 23). **Servidor** é a máquina que detém o sistema de redirecionamento de conexões, e **cliente** é a aplicação Java que requisita a conexão a um serviço específico. Apesar de não ser excludente, considera-se que normalmente o serviço, o servidor e o cliente estão sendo executados em máquinas diferentes.

Esse redirecionamento traz mais uma vantagem à segurança, e mais um diferencial dos outros sistemas de cifragem de transmissão (por exemplo, o *ssh*): havendo apenas um servidor de redirecionamento, este único servidor pode atender requisições para diversos serviços, desde que usem o mesmo tipo de *sockets* TCP (no Java estão disponíveis diversos "estilos" de *sockets*, como o bufferizado, utilizados de acordo com a necessidade da aplicação). Como somente esse servidor necessita ficar visível para toda a rede, pode-se ainda implementar um sistema de *firewall*, o que aumenta a segurança da rede interna e restringe o acesso aos serviços internos (o acesso passa a ser obrigatório através do servidor de redirecionamento). A Figura 5 demonstra como se pode gerenciar as requisições de serviços da rede usando um *firewall*, sem prejuízos à segurança interna.

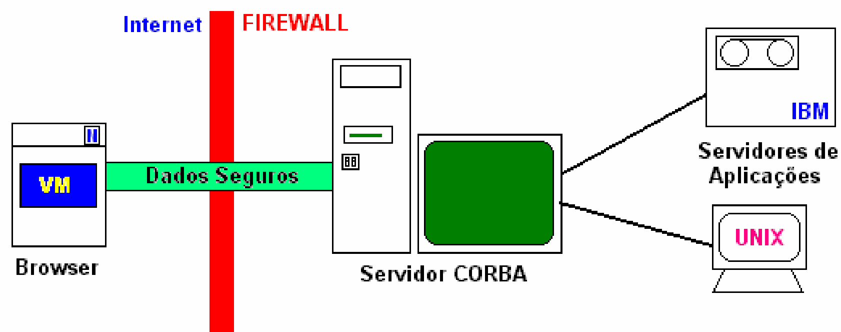


Figura 5 _ Segurança Através de *Firewall*

A conexão entre a aplicação no cliente (processo cliente) e o servidor de redirecionamento (processo servidor) é realizada utilizando-se a comunicação via o modelo CORBA.

O uso do CORBA permite uma maior abstração, no que se refere à busca e conexão de um serviço remoto, e à forma como são declarados os dados a serem enviados. Pode-se enviar estruturas complexas de dados sem preocupações quanto ao *Marshalling* (compatibilização de dados enviados entre máquinas de arquiteturas diferentes), localização e contato entre processos, etc., pois o CORBA realiza esse tratamento automaticamente, através da relação das definições em IDL. [VOG 97a].

Enquanto em uma arquitetura de Cliente-Servidor convencional do CORBA costuma utilizar um único processo servidor atendendo às requisições de diversos clientes, a arquitetura proposta exige cuidados especiais, tais como a manutenção permanente da conexão entre cliente-servidor-serviço. Para isso, escolheu-se um sistema alternativo, disponibilizado pelo CORBA, onde é designado um processo servidor para cada cliente. Essa forma de tratamento de requisições tem o nome de *Factory Service*. Nele, a cada nova conexão de um cliente, um processo servidor é criado e fica respondendo apenas àquele cliente. Quando o cliente termina suas atividades, ao fechar a conexão, o processo servidor a ele designado é eliminado. O processo de redirecionamento das conexões (e das mensagens) é mostrado na Figura 6.

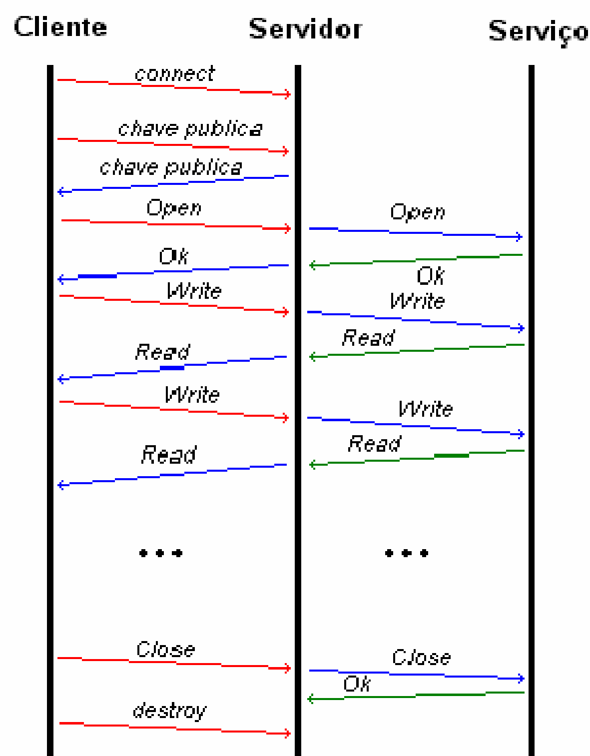


Figura 6 _ Seqüência de Mensagens entre Cliente-Servidor-Serviço

Para tornar transparente ao programador os detalhes da comunicação entre os processos cliente e servidor, o CORBA possui suas classes próprias, que permitem contactar (ou criar) um ORB local, e através dele comunicar-se com o ORB do servidor (essa comunicação envolvendo cliente-ORB-servidor é o chamado *three-tier*).

Através da definição do padrão CORBA 2.0 [VOG 97a], não há incompatibilidade na comunicação entre dois ORBs de fabricantes diferentes, o que possibilita uma total comunicação entre ORBs, com o uso do IIOP (*Internet Inter-ORB Protocol*). No entanto, o mesmo padrão CORBA 2.0 não definiu rigidamente as interfaces de inicialização, operação e atributos de execução, deixando a cargo de cada fabricante escolher entre diversas possibilidades quais ele mais se adapta. Isso acarretou que um sistema escrito usando as classes de um fabricante dificilmente consegue conectar-se diretamente a ORBs de fabricantes diferentes, embora ambos ORBs sejam capazes de intercomunicação completa. A possível solução para esse problema de incompatibilidade está no padrão CORBA 2.3, que define melhor os modelos de inicialização e execução dos ORBs, mas não foi possível testar nenhum ORB que já a implementasse.

Especificamente no modelo proposto, foram utilizadas classes do ORB *ORBacus*, da *Object Oriented Concepts Inc.*, na programação das interfaces do cliente e do servidor. *Browsers* que não dispõe de um ORB interno (por exemplo, o *Internet Explorer 4* e o *appletviewer*, que foram usados em testes) buscam as classes necessárias junto ao servidor de origem do *applet*, e executam as aplicações normalmente. O *Netscape Communicator*, entretanto, possui internamente as classes do ORB criado pela *Visigenic Inc.*, e não é possível executar as aplicações apropriadamente.

Assim, para utilizar uma das aplicações desenvolvidas utilizando a arquitetura aqui descrita em um *browser* com ORB interno, como o *Netscape Communicator*, existem duas possibilidades: a primeira é um artifício usado pelos programadores, onde são redefinidas as classes do ORB através de parâmetros na inicialização do *applet* [OOC 98], e a segunda através da eliminação das classes do ORB do *Netscape*. Infelizmente a primeira solução não é possível se for considerado um dos objetivos principais do sistema, que é o de assegurar uma comunicação segura a aplicações já existentes, sem mudanças drásticas na sua estrutura interna. No caso das aplicações de *telnet* e *TN3270* testadas, essa modificação representaria uma alteração em 3 ou 4 níveis de hierarquia das classes. Mas esse artifício ainda pode ser usado caso sejam desenvolvidas aplicações já tendo como base as classes de comunicação segura.

A segunda forma, mais recomendada e eficiente para este trabalho, envolve a eliminação da biblioteca de classes do ORB do *Netscape*, presente no diretório de

instalação. O arquivo que reúne as classes desse ORB chama-se *iiop10.jar*, que nas arquiteturas Windows95/98/NT normalmente é encontrado instalado no diretório *C:\Arquivos de Programas\Netscape\Program\java\classes*. Ao remover esse arquivo, o *Netscape* é obrigado a utilizar as classes do ORB fornecidas pela aplicação. A eliminação desse arquivo não traz desvantagens aparentes, pois não remove nenhum outro item essencial ao funcionamento do *Netscape* ou do reconhecimento do Java nesse *browser*.

Ao alcançar a compatibilidade necessária para a execução do cliente nos *browsers* mais populares, deve-se prestar atenção à criptografia, que é a principal responsável pela segurança dos dados transmitidos, sendo utilizada no modelo proposto tanto para a cifragem da mensagem quanto para assinar sua origem. O método criptográfico escolhido para a implementação do sistema foi o RSA, baseado em chaves públicas e privadas, e que possibilita as operações de cifragem e assinatura através de operações similares.

As chaves criptográficas utilizadas na comunicação são geradas apenas uma vez, durante a inicialização dos processos cliente e servidor (Figura 7). O fato das chaves serem geradas apenas uma vez a cada conexão não acarreta nenhum risco aos dados transmitidos, uma vez que o sistema criptográfico é totalmente seguro no decorrer do tempo de uma conexão, devido ao uso de outras medidas de proteção, tais como a autenticação e o *timeout*, discutidos mais adiante.

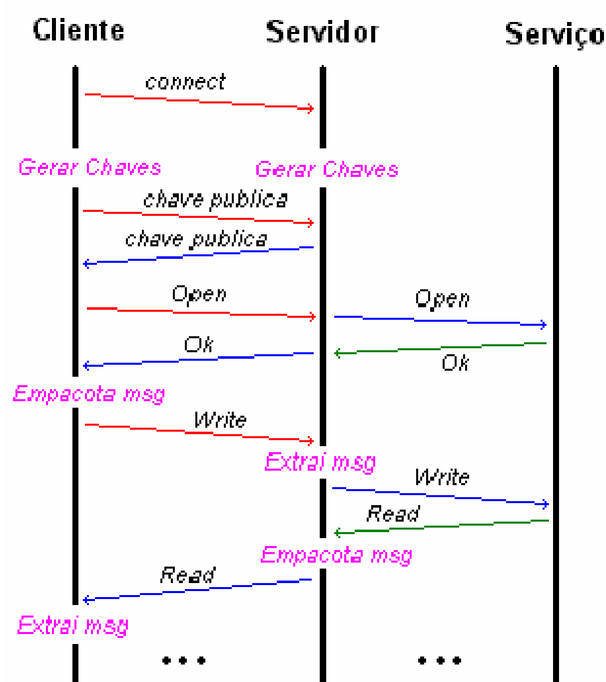


Figura 7 _ Aspectos da Comunicação Cifrada

A geração das chaves pública e privada, apesar de simples, demanda um esforço de processamento, e por conseguinte, tempo, de acordo com o nível de segurança das chaves criptográficas desejadas (nível esse correspondente ao comprimento das chaves empregadas, para o RSA).

O emprego de chaves consideradas virtualmente inquebráveis (1024 ou 2048 bits, para o RSA) é totalmente possível, mas acarreta um tempo de geração de chaves e cifragem muito grande. No Java esse tempo de geração é agravado devido à linguagem ser interpretada, exigindo cerca de 20 a 70 segundos para gerar chaves desses comprimentos, em uma máquina Pentium 200 Mhz. Além disso, soma-se o tempo de *download* das classes da aplicação e do ORB, o que pode acarretar um descontentamento do usuário, devido ao chamado *Response Time*.

Para evitar essa demora muito grande, pode-se considerar suficientemente confiável o emprego de chaves entre 256 e 512 bits, que apresentam um tempo de geração de chaves em torno de 2 a 5 segundos. Chaves com esse tamanho já foram quebradas, mas o tempo que demorou (8 meses usando 1600 computadores conjuntamente [RSA 98]) ainda as torna suficientemente seguras para as conexões usuais na rede.

Uma vez geradas as chaves, os processos cliente e servidor trocam suas chaves públicas (Figura 7), que serão usadas nos processos de cifragem e aferição da assinatura da outra parte. Não há risco em trafegar as chaves públicas, pois seu propósito é justamente oferecer a facilidade de propagação sem riscos à segurança do sistema criptográfico.

Ao requisitar uma conexão de rede, o cliente comunica ao processo servidor a máquina destino e sua porta de comunicação. O servidor então cria uma comunicação *socket* tradicional até o serviço. Não é possível manter a conexão segura entre o processo servidor e o serviço, porquê o serviço não está preparado para receber mensagens cifradas, e não objetiva-se modificar o serviço. Entretanto, supõe-se que o servidor e o serviço estejam interligados através de uma rede confiável, havendo então um risco muito menor sobre o sigilo dos dados.

Uma vez conectado, o servidor mantém a conexão ativa até que ocorra uma requisição explícita de desconexão, por parte do cliente. Isso não é o suficiente, pois

deve-se também tratar as eventuais quebras de conexão originadas por parte do serviço ou do servidor.

Através dessa conexão, o servidor repassa as mensagens entre o cliente e o serviço contratado, processando os dados de forma a serem transmitidos a cada uma das partes no formato exigido.

O tratamento das mensagens transportadas entre o cliente e o servidor inclui a criptografia, a assinatura digital, a geração de *tickets* e a adequação da mensagem para o transporte, através do fluxo esquematizado na Figura 8.

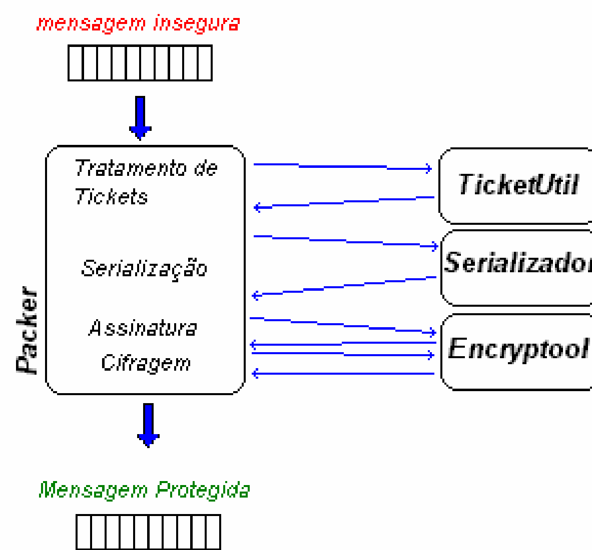


Figura 8 _ Fluxo do Processamento da Mensagem

O primeiro procedimento adotado determina a geração de *tickets* identificadores da transmissão. Um *ticket* é composto por um número aleatório e um valor de tempo, chamado *timestamp*, que garantem uma identificação única do *ticket* em toda comunicação.

Os *tickets* do servidor e do cliente são então anexados à mensagem. A necessidade de dois *tickets* deve-se à decisão de assegurar tanto a identificação única de cada transmissão (de fato, a cada transmissão de dados de uma das partes é gerado um novo *ticket*), quanto limitação de tempo máximo para a ociosidade da conexão. Essa limitação é obtida comparando-se o valor de tempo atual com o valor armazenado no *timestamp* da última transmissão.

Uma vez anexados a mensagem e os *tickets* do cliente e do servidor, eles serão **serializados**, ou seja, decompostos em um conjunto de informações descritivas da sua estrutura e conteúdo, e armazenados em um *array* de *bytes*.

Essa decomposição por si só oferece a capacidade de transmitir estruturas e objetos complexos através de uma rede, e reconstituí-los sem a necessidade de especificar protocolos específicos para uma estrutura. Um objeto serializado pode ser enviado normalmente através de uma conexão *socket*.

Uma vez serializado, o pacote que contém os *tickets* e a mensagem ainda é submetido a dois processos criptográficos. O primeiro é a assinatura digital, que cifra o *array* de *bytes* com a chave privada do remetente, indentificando a origem da mensagem, e por último o processo de criptografia, que se utiliza da chave pública do destinatário, como forma de garantir que somente ele seja capaz de decifrar a mensagem recebida. As camadas de segurança são representadas na Figura 9.

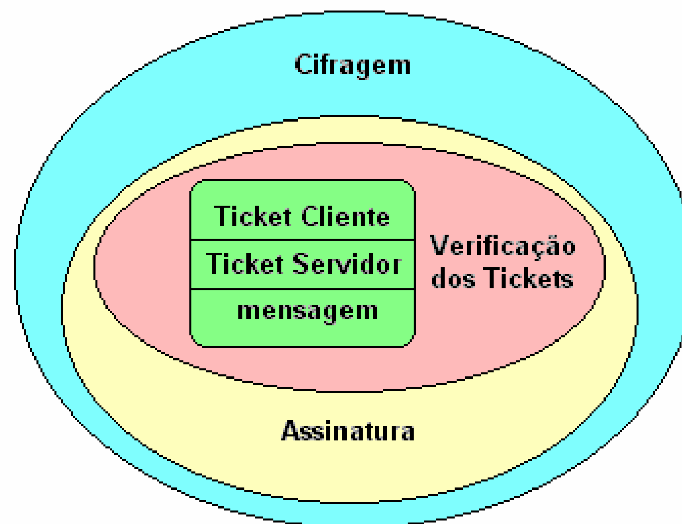


Figura 9 _ Camadas de Segurança sobre a Mensagem

No lado do receptor, ocorrem os processos contrários: a mensagem recebida é decifrada, usando a chave privada do receptor. Essa mensagem decriptada é então enviada à operação de verificação da assinatura utilizando-se da chave pública do remetente, obtida no início da conexão.Se a identificação de origem for inválida, não será possível recompor os elementos da estrutura que transporta os dados, e portanto, extrair a mensagem.

Se a verificação da assinatura for correta, no próximo passo a desserialização irá remontar o pacote de dados que contém os *tickets* e a mensagem.

Antes de acessar a mensagem, ainda é verificado o *ticket* correspondente ao receptor, que é comparado ao *ticket* original ainda armazenado. Se for aceito, é verificada a validade do *timeout*, e caso obtenha sucesso, o *ticket* do emissor é armazenado para ser incluído na próxima transmissão. Somente então a mensagem original é extraída do objeto onde estava encapsulada.

3.2 Tickets

Uma das ferramentas utilizadas para garantir a autenticação da mensagem, junto com a assinatura digital, é o uso de *tickets* para identificar uma transmissão.

Os *tickets*, segundo o conceito utilizado, são identificadores únicos de cada conexão. Cada parte envolvida tem seu *ticket*, que serve tanto para a detecção da ordem de envio (evitando retransmissões), quanto para a determinação do instante da transmissão. Para isso, considera-se a associação de dois valores: um número identificador (randômico) e uma assinatura de tempo (*timestamp*). Esse par garante uma identificação única a cada *ticket*, no período da conexão.

O número identificador usado é uma seqüência de 30 *bytes*, gerados randomicamente. A decisão de utilizar essa combinação de números, ao invés de um único valor, deve-se a algumas restrições que o Java faz quanto à geração de números randômicos, a ponto de não classificar assim a geração de *bytes* como “pseudo-randômicos”,ao contrário da geração de valores inteiros e de ponto flutuante. Além disso, a geração de vários *bytes* ao invés de um único tende a fugir do risco de uma periodicidade na distribuição dos valores (principalmente porque em uma seqüência de 30 *bytes*, a ordem também é um fator importante).

Já o *timestamp* é um valor numérico que representa a data e a hora local da máquina, medido em milissegundos (padrão do Java) a partir de 1º de janeiro de 1970, obtido a partir das primitivas do sistema.

Um *ticket*, então, é a reunião desses dois conjuntos de valores. Para facilitar a abstração sobre o conceito de *ticket*, é interessante reunir os valores em um único

objeto. A decisão habitual seria gerar um objeto do tipo Ticket, mas devido ao Java já prover um tipo de objeto “container” (o objeto *Vector*, uma classe que permite a anexação de objetos de tipos diversos, como numa lista), foi julgado não ser necessário implementar um novo tipo de objetos.

A classe *Vector* é uma lista polimórfica de objetos, ou seja, aceita que quaisquer tipos de objetos sejam a ele ligados. Essa estrutura torna necessário que os dados componentes de um *ticket* sejam tratados antes de encapsulá-los no *Vector*. O *array* de *byte* é facilmente transformado em um objeto do tipo *String*, de acordo com as próprias definições da API do Java [JDK 97], e o inteiro longo que contém o valor do *timestamp* também é transformado em objeto utilizando-se uma *wrapper class* específica. Uma *wrapper class* é uma das diversas classes de objetos implementadas pelo Java que mantém funcionalidades semelhantes aos tipos numéricos primitivos, onde por exemplo o tipo *long* pode ser transformado no objeto *Long*, como mostra a Figura 10.

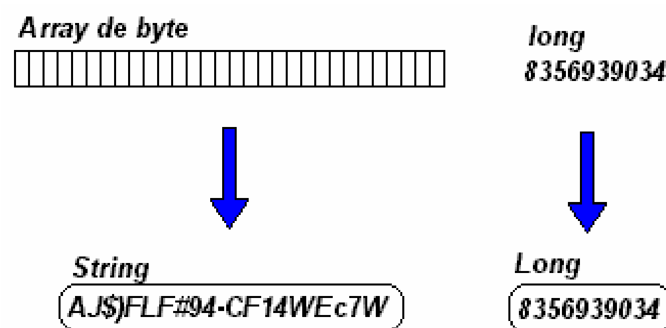


Figura 10 _ Encapsulamento dos Tipos Primitivos

Representados por objetos equivalentes, os dados podem ser anexados à estrutura do *ticket*, que irá representar a identificação de uma das partes (cliente ou sevidor) a cada troca de informações (Figura 11).

O motivo de manter dois *tickets*, um para cada parte, é devido ao processo de autenticação e *timeout*. Quando uma mensagem é recebida, primeiro é verificado o *ticket* referente ao receptor. A verificação do *ticket* é feita em duas etapas: na primeira, são comparados os números identificadores do *ticket* recebido e do *ticket* armazenado no servidor desde a última transmissão. Essa comparação foi implementada a nível de *String* (essa *String* é a representação da seqüencia de *bytes*,

quando foram encapsulados), não havendo a necessidade de extrair os números, pois há uma equivalência entre os *bytes* e a *String* que foi gerada.

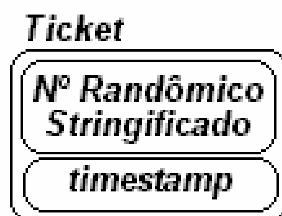


Figura 11 _ Estrutura Interna de um *Ticket*

Quando essa comparação retorna uma indicação positiva, é verificado ainda o tempo decorrido desde a última transmissão (*timeout*). Para isso, extrai-se os valores numéricos que representam o tempo, e é feita a subtração para obter a diferença, que irá representar o tempo transcorrido, em milissegundos. Se esse valor de tempo for superior ao máximo permitido (por definição, foi escolhido o valor genérico de 15 minutos), o *ticket* não é aceito e o sistema deve tratar essa situação fechando a conexão com o serviço (embora a implementação dessa medida de segurança careça de uma melhor resposta ao usuário).

Se for aceito, o *ticket* do emissor é então armazenado, para a próxima transmissão, e a mensagem pode ser lida. Não é gerado um novo *ticket* do receptor ainda, porque pode ocorrer uma situação em que várias mensagens são recebidas sem que ocorra nenhum envio. Isso não compromete o sistema, pois a combinação dos dois *tickets* continua gerando a identidade única da transmissão. A geração de um novo *ticket* se dará no momento de uma nova transmissão de dados.

Esse tipo de proteção é eficaz para evitar que uma mensagem, após ser decodificada por um indivíduo não autorizado, possa ser reinserida na comunicação. Como usualmente são trocadas várias mensagens em poucos segundos, evita-se que o atacante se torne um personagem ativo na comunicação, inserindo dados forjados.

No entanto, caso a comunicação fique ociosa por alguns instantes, um atacante poderia tentar decifrar a mensagem, para reenviá-la no lugar de uma das partes, tomando seu lugar na conexão sem autorização. A checagem do *timeout* garante que o atacante não terá tempo suficiente para fazer isso, desde que o valor limite de tempo seja inferior ao tempo mínimo para a quebra da criptografia que protege a mensagem.

Uma possível oportunidade de falha do sistema de proteção ocorre quando da criação dos *tickets* da mensagem inicial. Como sempre há a checagem dos *tickets*, houve a necessidade de sincronizar o cliente e o servidor com valores de *tickets* conhecidos por ambas as partes. Esse risco pode ser reduzido ao modificar os valores iniciais para ambos servidor e cliente em tempo de execução, a cada nova requisição de conexão, por exemplo.

3.3 Serialização

Seguindo o processo do tratamento da mensagem para torná-la apta a trafegar pela rede em segurança, devem ser reunidos os *tickets* identificadores da transmissão (gerados pelo cliente e pelo servidor), e a mensagem em uma estrutura única, um objeto, de modo a facilitar a abstração da transmissão dos dados. Esse objeto que contém os dados foi implementado utilizando a classe *Vector* definido na linguagem Java, que suporta a anexação de objetos diferentes em uma mesma instância de *Vector*, agindo como uma lista de elementos.

Uma vez reunidos os *tickets* e a mensagem, deve-se efetuar a assinatura e a encriptação. O método criptográfico empregado não define uma forma de cifrar diretamente um objeto, o que torna obrigatória a conversão desse objeto para uma notação aceita pela função criptográfica. O formato de dados aceito pelas interfaces de cifragem RSA, definidos pelos métodos *encrypt* e *decrypt* da classe *Key*, definida pela biblioteca *Cryptonite*, restringe-se a *arrays* de *bytes*. Isso é compatível com a maior parte do emprego tradicional da criptografia (cifragem sobre um documento, normalmente composto apenas por caracteres). Desse modo, objetiva-se determinar um método simples de converter um objeto em um *array* de *bytes*, mantendo a consistência dos dados, para posterior reconstituição.

A linguagem Java define um método interessante para muitas aplicações, mas que é especialmente compatível com a necessidade do sistema. Esse método é chamado de **serialização de objetos** (cuja utilização é descrita em [JDK 97] e [JDC 98]), e implementa uma maneira de descrever um objeto em sua estrutura e conteúdo, e a forma de construir um objeto exatamente igual ao que foi descrito.

Utilizando-se os exemplos contidos em [JDK 97], foi criada uma classe chamada *Serializador*, que implementa uma interface de serialização e desserialização (Figura 12), específica para a conversão das classes *Vector* que contém os *tickets* e a mensagem, em uma estrutura do tipo *array* de *bytes*.

```
byte[] serializar (Vector v) {
        ByteArrayOutputStream bout = new ByteArrayOutputStream();
        ObjectOutputStream out = new ObjectOutputStream(bout);
        out.writeObject(v);
        byte arraya[] = bout.toByteArray();
        return arraya;
}

 Vector desserializar(byte source[]) {
       ByteArrayInputStream bin = new ByteArrayInputStream(source);
        ObjectInputStream in = new ObjectInputStream(bin);
        Vector vet2 = (Vector)in.readObject();
        return vet2;
 }
```

Figura 12 _ Serialização de um Objeto

A serialização descrita pela documentação da linguagem Java é uma implementação proprietária do conceito de representação de dados conhecida como ***Marshalling***. Segundo [BIR 95], *Marshalling* é um mecanismo de representar os dados de forma que possam ser correta e eficientemente interpretados pelo receptor da mensagem. Na maioria dos casos, esse mecanismo trata a possibilidade de existirem computadores clientes e servidores rodando em arquiteturas diferentes, e portanto, com um grande possibilidade de diferentes representações dos dados.

De fato, [BIR 95] cita que até existe um padrão para essa representação, o ASN.1, onde "objetos de dados auto-descritos" podem ser representados. Ainda assim, muitos fabricantes optam por produzir sua própria estrutura de representação, como o *External Data Representation* (XDR) da *Sun Microsystems*, usado no protocolo NFS _ *Network File System*.

Um dos principais problemas enfrentados por essa heterogeneidade das arquiteturas é o da representação dos números inteiros. Enquanto alguns processadores identificam como o bit mais significativo de um número inteiro o *bit*

que está localizado no início da palavra da memória, outros processadores consideram o *bit* que está no fim. Isso acarreta que em um processador, o número representado por 0101 seja reconhecido como o inteiro 5, e em outra arquitetura de processadores essa seqüencia de bits tenha o valor 10. Isso não pode ser resolvido simplesmente invertendo a ordem de todos bits, porque esse problema não afeta outros tipos de dados, como os caracteres [TAN 95]. Essas representações, chamadas de *little-endian* e *big-endian*, devem ser tratadas através da adição de uma camada descritiva do tipo dos dados e da arquitetura de origem.

Apesar do Java não ter esse problema com os números, pois todas máquinas virtuais Java (*Java Virtual Machine _ JVM*), onde rodam as aplicações, serem construídas em software com a mesma representação e tipos de dados, ainda há a necessidade de descrever a estrutura dos objetos e dados, como forma de facilitar a transmissão através da rede para outros clientes e servidores Java (Figura 13). Assim, a serialização foi definida para ser uma ferramenta de grande ajuda ao sistema de RPC _ *Remote Process Call* implementado na linguagem Java, o *Remote Method Invocation* (RMI) [JDK 97].

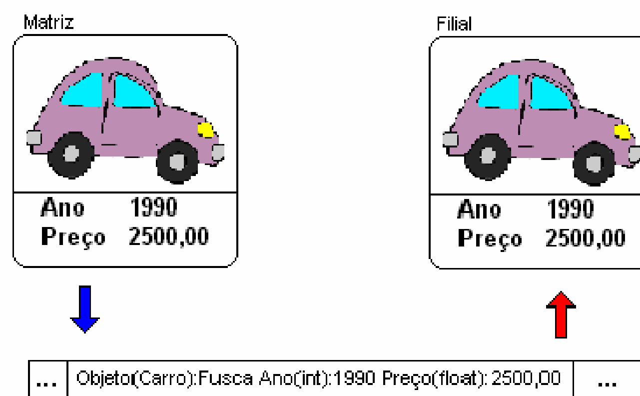


Figura 13 _ Serialização de um Objeto Fusca

Embora o método descrito aqui é exclusivo para a serialização de objetos do tipo *Vector*, é possível estender essa forma de representação para todos objetos. Isso, associado ao emprego dos métodos de criptografia do *Cryptonite* garantiriam uma forma rápida e simples de cifragem de objetos. Essa opção pode ser muito explorada em diversas áreas, indo desde a comunicação simples (por exemplo, através de *sockets* ou do *RMI*), até outras formas de utilização de objetos serializados, tais como

um banco de dados contendo objetos serializados com acesso restrito aos seus proprietários (tais como um cofre de jóias em um banco), entre outras.

3.4 Processos Criptográficos – Assinatura Digital e Cifragem

Como já foi descrito na Seção 3.1, referente à Análise Geral, o sistema deve se encarregar de transportar os dados requisitados pela aplicação através das chamadas aos métodos *socket-like* (semelhantes aos dos *sockets*), definidos na classe cliente *CSClient*. Esses dados trafegarão através da rede utilizando as facilidades CORBA, e serão recebidos pela classe servidora *CSServer_impl*. Para realizar as operações criptográficas de assinatura e cifragem exigidas para a comunicação segura dos dados, foram definidas classes que realizam os processos sobre os dados serializados, como forma de inserir mais um item de autenticação e conferir a privacidade necessária aos dados transportados através de uma rede insegura.

Os processos criptográficos, então, compreendem a mais importante medida de segurança para garantir a inviolabilidade dos dados trafegados. A utilização dos processos de cifragem e assinatura, conjuntamente, visa proteger os dados tanto de ataques ao processo emissor quanto ao processo receptor.

A escolha do uso de chaves criptográficas assimétricas (chaves públicas e privadas), especificamente o sistema RSA, permite que, através de uma operação semelhante, os dados possam ser cifrados ou assinados digitalmente, bastando apenas mudar a chave de encriptação.

Nos sistemas de criptografia de chave pública, qualquer pessoa pode cifrar uma mensagem, mas somente o destinatário desta mensagem pode decifrá-la. Isto é obtido justamente pelo uso de duas chaves: uma pública, para cifragem, disponível para qualquer um, e outra privada, para decifragem, conhecida apenas por uma pessoa. Mensagens cifradas pela chave pública somente poderão ser decifradas pela chave privada correspondente.

Invertendo a ordem de uso das chaves, obtém-se uma mensagem que só pode ser cifrada por uma pessoa, mas pode ser decifrada por qualquer um. Naturalmente, não se pode falar em privacidade ou segredo neste caso, mas obtém-se um efeito de

personalização do documento (somente uma pessoa pode tê-lo cifrado), semelhante a uma assinatura. Um sistema deste tipo é denominado de assinatura digital.

Para a geração de chaves, deve-se escolher um comprimento de chave adequado às requisições de segurança e tempo de processamento necessários. A Tabela 4 mostra os valores obtidos em um teste, para a geração de chaves entre 64 e 2048 bits, em um computador Pentium 200Mhz. Considerando a possibilidade do servidor de redirecionamento estar atendendo muitas requisições, esses valores podem ser maiores, levando a uma sobrecarga excessiva do servidor. Uma sobrecarga no servidor leva o sistema a um gargalo, que compromete seriamente o já complexo processo de cifragem dos dados.

Segundo a Tabela 4, obtida com o auxílio da aplicação de testes *TestKeyGeneration*, do pacote *Cryptonite*, pode-se afirmar que comprimentos de chave com menos de 1024 bits são suficientemente rápidos. E de acordo com [RSA 98], valores na faixa entre 256 e 512 bits são suficientemente seguros, no tempo esperado de duração de uma conexão, ou seja, de alguns minutos até algumas horas.

Tabela 4 _ Tempo de geração de chaves RSA

Comprimento da Chave (bits)	*Tempo de Geração (segundos)*
64	0.930
128	0.940
256	1.970
512	4.340
1024	18.680
2048	74.310

Ao requisitar um servidor que o atenda, o cliente dispara a criação de suas chaves criptográficas, bem como do servidor recém criado (são designados servidores dinamicamente, através do uso de um *Factory Server*, descrito na Seção 4.2). As chaves geradas pelo método *make_key* da classe *Encryptool* são armazenadas na classe superior, *Packer*, que coordena as operações de cifragem e assinatura, o que facilita a troca de chaves públicas entre o cliente e o servidor. As chaves pública e privada vêm encapsuladas dentro de um objeto do tipo *KeyPair*, definido pelo pacote *Cryptonite*. Essas chaves são acessadas pela classe *Packer* através dos métodos

KeyPair.getPublic() e *KeyPair.getPrivate()*, e são enviadas aos métodos de cifragem para a assinatura e a encriptação.

À geração das chaves criptográficas do sistema foi adicionada ainda uma outra medida de segurança, para incrementar a diversidade de chaves e reduzir a possibilidade de quebra por ataque do texto cifrado (Seção 2.1.2). Quando uma chave é gerada, são fornecidos como parâmetros obrigatórios o nome e o e-mail do proprietário, como forma de identificar o proprietário (em um sistema criptográfico tradicional, uma chave gerada é utilizada diversas vezes pelo usuário). Neste sistema não há a necessidade de um uso direto desses parâmetros, pois as chaves são geradas apenas para uma sessão e depois descartadas. Assim, para dar uma maior diversidade às chaves geradas, são fornecidos valores randômicos para os campos de nome e e-mail, ao invés dos dados de um único usuário (real ou não).

Outro fator muito importante a ser notado, em uma aplicação cuja troca de mensagens é muito freqüente, é o *overhead* representado pelas operações de cifragem e decifragem. Além da demora inicial para a criação das chaves, a cada operação criptográfica sobre a mensagem (e este sistema implementa duas operações, cifragem e assinatura) há um tempo de processamento assimétrico, ou seja, a operação de cifragem exige um esforço bem menor do que a de decifragem. Utilizando outra aplicação de testes do pacote *Cryptonite*, a classe *TestEncryption*, obteve-se os seguintes parâmetros, para a cifragem e decifragem de um bloco de 2048 bits aleatórios. É importante considerar esse tamanho de bloco. Levando em conta que um emulador *TN3270* opera enviando e recebendo páginas inteiras para a tela, e calculando 25 linhas x 80 colunas, já são 2000 bytes, mais os bytes que representam comandos, facilmente chega-se próximo aos valores apresentados na Tabela 5.

Tabela 5 _ Tempos de Cifragem e Decifragem

Comprimento da Chave (RSA)	*Tempo de Cifragem*	*Tempo de Decifragem*
128 bits	0.60s	0.380s
256 bits	0.50s	1.150s
512 bits	0.110s	3.790s
1024 bits	0.110s	14.10s

2048 bits	0.270s	55.970s

O uso de chaves maiores de 512 bits acarretam um *delay* consideravelmente grande, que embora seja aceitável para a cifragem de documentos estáticos, é inaceitável em aplicações onde há uma um grande número de mensagens circulando e o contentamento do usuário é diretamente ligado à velocidade da resposta.

Na implementação, as chaves geradas são representadas por objetos *Key*, definidos no pacote *Criptonite*, e armazenadas dentro de um outro objeto do tipo *KeyPair*. Para comandar as operações de cifragem e assinatura, bem com as operações opostas a elas, a classe *Packer* extrai a chave pública ou privada desejada através dos métodos *KeyPair.getPublic()* e *KeyPair.getPrivate()*, do próprio objeto que contém as chaves. A chave pública é extraída apenas para ser enviada à outra extremidade da conexão, e será armazenada por uma instância da classe *Packer* do outro lado. Podendo acessar as três chaves (pública e privada dela própria, e a chave pública do outro extremo), a classe *Packer* repassa-as junto com a mensagem aos métodos *encrypt* e *decrypt* da classe *Encryptool*, de acordo com os seguintes objetivos, descritos na Tabela 6:

Tabela 6 _ Operações sobre as Chaves

Operação	**Chave**
Cifragem	Pública externa
Decifragem	Privada local
Assinatura	Privada local
Validação da Assinatura	Pública externa

Os métodos de cifragem e decifragem são internos às próprias chaves, de forma que sua utilização equivale à descrita na Figura 14.

```
public byte[] encrypt(byte[] message, Key chave) {

        byte dest[] = chave.encrypt(message);        // Cifragem (ou assinatura)
        return dest;
}

public byte[] decrypt(byte[] message, Key chave) {

        byte dest[] = chave.decrypt(message);        // Decifragem (ou verificação)
        return dest;
}
```

Figura 14 _ Operações de Criptografia

A implementação do sistema utiliza a versão 0.08 do pacote *Cryptonite*. Apesar do número de versão, essa distribuição é bem estável e com diversas opções. Atualmente, a versão é a 0.53, que incorporou outros métodos de cifragem e de *hash*. No entanto, não foi constatada nenhuma otimização sobre o método RSA, e as interfaces de acesso às classes de criptografia foram alteradas. Para evitar ter que readaptar as classes já construídas foi decidido continuar o uso da versão 0.08.

A demora apresentada na geração das chaves era um valor esperado, mas os valores obtidos na cifragem e na decifragem não eram esperados (principalmente a diferença entre os processos). Para solucionar isso, uma possível implementação seria a empregada no *ssh*, ou seja, usa-se chaves RSA apenas para transmitir as chaves DES que serão empregadas no decorrer da transmissão. O uso do DES ou de suas variantes (Triple-DES, etc.) deve aumentar a velocidade do sistema; embora não exista um teste realizado para as classes em Java, sua performance deve ser comparável à predita por [RSA 98], ou seja, cerca de 100 vezes mais rápido que o RSA.

O emprego de tal método acarreta alguns problemas, pois uma das formas de autenticação (a assinatura digital) terá que ser substituída por uma variante, talvez uma assinatura por *hash*.

3.5 Análise de Interfaces

As interfaces são as chamadas por onde o CORBA permite o tráfego de informações entre o cliente e o servidor, de forma a parametrizar os tipos de dados de cada acesso.

O CORBA possui uma Linguagem de Definição de Interfaces (IDL), onde são descritas as interfaces utilizadas e os dados por elas trafegados. Com base em uma definição IDL, o pré-processador de IDL monta uma estrutura de classes que coordenam o CORBA no envio e recepção dos dados, tanto no lado do servidor quanto do cliente. Assim, são criados os *stubs* (classes no lado do cliente que fazem a requisição remota ao servidor) e os *skeletons* (classes apenas com a estrutura dos métodos, que serão preenchidos com a implementação do servidor).

Para a definição das interfaces a serem utilizadas, há a necessidade de detectar quais são as vias de comunicação entre o cliente e o servidor, bem como o formato dos dados por elas trafegado. Assim, três áreas devem ser estudadas, para a definição das interfaces: a comunicação no estilo *socket*, as operações de segurança implementadas e o gerenciamento do *Factory Server*.

3.5.1 Interfaces Socket

A definição das intefaces CORBA utilizadas na comunicação entre os clientes e os processos servidores de redirecionamento baseou-se nas principais chamadas aos métodos de uma comunicação *socket* tradicional.

Existem muitos métodos disponíveis nos *sockets* da linguagem Java. Muitos desse métodos não tem relação direta com o simples envio dos dados (por exemplo, o método *setSocketImplFactory*, usado para a definição de um *Factory* puramente de *sockets*). Também existem métodos cuja implementação é difícil em um ambiente de redirecionamento, como, por exemplo, a obtenção do endereço IP das “pontas” da conexão. Adaptar esse tipo de método exige um trabalho não relacionado diretamente com os objetivos deste trabalho.

Ao optar-se por uma arquitetura de redirecionamento, é menor a necessidade de disponibilizar vários servidores em máquinas diferentes, uma vez que basta um para atingir toda rede, então foi escolhido o formato de referência estática para a conexão entre o cliente e o servidor. No modo de conexão estática, a *string* que identifica o servidor de forma única é distribuída diretamente ao cliente, não utilizando nenhum artifício dinâmico de referência (por exemplo, um *Naming Directory*, um servidor que cadastra referências e as repassa de acordo com a necessidade do cliente). A forma

escolhida para disponibilizar esse IOR (*Interoperable Object Reference)* foi através de um arquivo disponibilizado no mesmo endereço Internet do servidor.

Ao inicializar, o servidor gera esse arquivo (nesse trabalho, denominado *CsocketS.ref*), e o cliente busca esse arquivo antes de realizar uma conexão com o servidor.

Para as interfaces, uma análise dos métodos utilizados nas duas aplicações de teste (*telnet* e *TN3270*) determinou como mínimo a ser implementado as seguintes interfaces de comunicação:

- *Open*;
- *Close*;
- *Read*;
- *Write*;
- *Available*.

As duas primeiras operações referem-se à criação e ao desligamento de uma conexão *socket*. As operações de *read* e *write* referem-se à transmissão de dados entre o cliente e o servidor, e a chamada *available* é destinada à negociação sobre a quantidade de dados que restam ser transmitidos.

A chamada *open* recebe como parâmetros o endereço e a porta onde é disponibilizado o serviço requisitado. Estes parâmetros são repassados ao servidor através da interface CORBA para que este crie e mantenha uma conexão *socket* tradicional com o serviço especificado.

A chamada *close* não contém nenhum parâmetro, apenas determina o fechamento da conexão.

Como *open* e *close* são utilizados apenas uma vez em todo o processo de comunicação, e contêm apenas informações básicas de referência a um *host* remoto (nome do *host* e porta do serviço), não há a necessidade de criptografar essa transmissão.

As operações que tratam da transmissão e recepção das mensagens cifradas, *read* e *write*, também baseia-se na implementação padrão dos *socket*.

Na primeira, é designado um *buffer*, para ser preenchido com os dados, retornando o número de *bytes* depositados no *buffer*. A interface do *write* deve prover a transmissão assíncrona de uma sequência de *bytes* para o destino.

O tratamento às mensagens deve ser implementado nas duas extremidades da transmissão, para garantir a segurança dos dados.

A última interface baseada nos *sockets* é a *available*. Essa interface é utilizada na implementação da aplicação de *telnet*, e indica quantos *bytes* ainda restam no *buffer* do emissor, após a leitura de um bloco. Isso é muito útil para o *telnet*, pois sua implementação normalmente recebe poucos *bytes* de cada vez, e há a necessidade de saber se há mais dados esperando a leitura.

3.5.2 Operações de Segurança

Devido à comunicação segura já ser executada através das interfaces *read* e *write*, restou apenas a necessidade de definir a interface para a troca das chaves públicas entre o cliente e o servidor.

O cliente envia sua chave pública para o servidor, que retribui mandando a sua chave pública, utilizando a interface *getKey*. Através dessa interface são transmitidas ambas as chaves, utilizando para isso de um *buffer* comum, que leva a chave pública do cliente num primeiro momento, e traz a chave do servidor em seguida.

3.5.3 Factory Server

Um *Factory Server* é um processo servidor que ao receber conexões de clientes, cria e designa um servidor específico para cada cliente. De fato, no momento de inicialização do servidor, somente o *Factory Server* é executado, e de acordo com as requisições dos clientes, novos objetos do tipo *CSServer_impl* (servidor) são instanciados e executados. Essa característica é muito útil para o sistema desenvolvido, pois a manutenção de uma conexão *socket* com um serviço exige de cada servidor uma total fidelidade e atenção ao cliente.

A implementação de um *Factory Server* exige que além da comunicação com o servidor de redirecionamento, exista também a comunicação com o servidor *Factory*. Para isso, é gerada uma interface para esse servidor, contendo como chamadas apenas a requisição para a criação de um novo processo servidor. Através dessa chamada o novo processo é referenciado, e então repassado para a conexão do cliente.

O processo servidor gerado também requer a criação de uma chamada *destroy*, destinada a notificar o ORB que esse processo servidor já encerrou sua execução, e que pode ser eliminado. Por fim, uma referência ao número de processos servidores já criados, útil para *debug* ou controle, é obtido através da chamada *get_id,* que retorna esse número.

3.5.4 Mapeamento das interfaces IDL

O mapeamento das interfaces inicia com a definição dos "servidores" que recebem essas requisições. O primeiro a ser definido, *CsocketS* corresponde ao servidor de redirecionamento, e irá conter todas as chamadas relacionadas com a conexão a um serviço e às informações de segurança.

O segundo servidor corresponde ao *Factory Server*, que recebe a requisição para a criação de um novo processo servidor de redirecionamento.

Em IDL isso é definido conforme a Figura 15:

module CsocketS {	*Package* CsocketS
interface CSServer {	Servidor de redirecionamento
};	
interface ConnFactory {	Servidor *Factory*
};	
};	

Figura 15 _ Definição dos Servidores

Ao servidor de redirecionamento devem ser adicionados os métodos referentes à comunicação *socket*, à troca de chaves e à finalização do processo servidor.

Os tipos de dados pré-definidos pelo IDL incluem o suporte a inteiros e strings, que serão usados, mas não a um *array* de *bytes*. Esse suporte deve ser feito através da

declaração de tipos definidos pelo usuário (*typedef*), no início da descrição IDL [OMG 97].

Devido à proposta de um emprego geral para as classes, optou-se por descrever esse *array* sem um tamanho pré-especificado, o tipo *sequence*. A definição desse *typedef* de bytes (*octet*, em IDL) é visualizada na Figura 16, e seu mapeamento em Java é feito pelo pré-processador IDL, que gera as classes necessárias, utilizadas na implementação do sistema.

```
module CsocketS {
        interface CSServer {
                typedef sequence<octet> larray;
        };
        interface ConnFactory {
        };
};
```

Figura 16 _ Declaração de um *Array* de *bytes*

Devido ao IDL não suportar o retorno de tipos de dados complexos, deve ser feita uma forma de passagem de parâmetros por referência, a exemplo da linguagem C. No IDL, a identificação do tipo de parâmetro é feita através da anexação das palavras ***in*** ou ***inout***. No primeiro caso, isso indica que o dado será apenas enviado, como numa passagem de parâmetros por cópia. A utilização de *inout* determina que os parâmetros passados poderão sofrer modificações, que devem ser notificadas a que as enviou. De fato, para simular essa passagem por referência, o CORBA gerencia as modificações sobre as variáveis, mantendo a consistência entre os dados do cliente e do servidor [VOG 97a], através da geração automática de mais uma classe de objetos, dependente do tipo de dado. No caso desta implementação, o IDL gera uma classe de objetos *LarrayHolder*, que é semelhante à seqüência de *bytes larray*, mas tem capacidade de retornar o valor modificado. Essa classe *LarrayHolder* é usada na implementação em Java, ao invés de um *array*, inserindo-se o valor do *array* em um dos atributos de um objeto *LarrayHolder*.

Como determinado na definição de quais interfaces usar, as chamadas a *read* e *getKey* deverão suportar *inout*. As demais, exigem apenas o formato *in*. A definição das chamadas do servidor seguro podem ser visualizadas na Figura 17.

```
module CsocketS {
        interface CSServer {
                typedef sequence<octet> larray;
                void open(in string host, in long port);
                void close();
                long read(inout larray rarray);
                void write(in larray warray);
                long available();
                void getKey(inout larray keyarray);
                long get_id();
                void destroy();
        };
        interface ConnFactory {
        };
};
```

Figura 17 _ Declaração das Chamadas a CSServer

Uma atenção deve ser dada ao envio do tipo *string*. Apesar de aceitar o mapeamento do tipo *String* do Java nos casos usuais, o tipo *string* do IDL somente aceita caracteres ASCII, enquanto o do Java suporta Unicode. No caso da passagem do nome do *host* a ser conectado, não há problemas, mas pode haver complicação com outros usos. A padronização do CORBA versão 2.3 estende esse mapeamento para Unicode.

Por último, devem ser definidas as chamadas ao *Factory Server*. Essa chamada refere-se ao método que inicializa um novo processo servidor, e retorna sua referência. Observando a definição desse método na Figura 18, observa-se que ele retorna um objeto CSServer. Ao contrário do que se pode imaginar, isso não contraria as definições de retorno do IDL comentadas acima. Na verdade, não é retornado um objeto, e sim uma referência (afinal, o cliente não tem que ter o servidor do seu lado, apenas a referência) a um servidor, com a interface semelhante ao "tipo" de objeto CSServer descrito logo acima na definição IDL.

Ao rodar o pré-processador IDL sobre a definição acima, são gerados os *stubs* e *skeletons* da aplicação, e com esse elementos pode-se implementar o cliente ou o sevidor na linguagem desejada, bastando ter um pré-processador para aquela linguagem. Através da estruturação imposta pelo IDL, clientes e servidores implementados em linguagens diferentes devem se comunicar normalmente, usando os dados e interfaces descritas.

```
module CsocketS {
        interface CSServer {
                typedef sequence<octet> larray;
                void open(in string host, in long port);
                void close();
                long read(inout larray rarray);
                void write(in larray warray);
                long available();
                void getKey(inout larray keyarray);
                long get_id();
                void destroy();
        };
        interface ConnFactory {
                CSServer CreateConn();
        };
};
```

Figura 18 _ Declaração das chamadas a ConnFactory

4 MODELAGEM DA ARQUITETURA

4.1 Introdução

As classes desenvolvidas foram projetadas para estender as funcionalidades da comunicação *socket* padrão, adicionando características de autenticação e sigilo através do uso de *tickets* de sessão e de criptografia. Portanto, a primeira característica da arquitetura desenvolvida é a herança das funcionalidades dos *sockets*. Em cima dessas funcionalidades, que consistem na criação e manutenção de conexões, transmissão e recepção de dados, são desenvolvidas as operações que possibilitam o aumento da segurança.

Essas operações exigem a construção de uma estrutura de apoio, que consiste no conjunto de classes responsáveis pelo processamento das tarefas de cifragem e criptografia, através do uso associado de criptografia de chave pública e *tickets* identificadores, bem como pela transmissão dos dados.

Conforme o objetivo proposto, as classes são encarregadas também pelo redirecionamento das mensagem de forma transparente para o usuário e o serviço final, de modo a minimizar as alterações necessárias em um aplicativo já existente. A escolha da utilização de uma tecnologia de distribuição de objetos como o CORBA propicia o aumento das funcionalidades possíveis, ao permitir que os esforços se concentrem nas operações de segurança, e não nas peculiaridades da comunicação entre cliente e servidor. Para melhor utilizar as propriedades do CORBA, o sistema também impleta o serviço *Factory Server*, responsável pela administração de múltiplos clientes e conexões.

4.2 Definição das Estruturas CORBA através da IDL

Levando em conta as restrições impostas pelos sistemas utilizados, a criação de uma arquitetura de classes que seja flexível e compatível com as interfaces *sockets* exige algumas especificações. A primeira refere-se à forma como serão conectados os clientes aos serviços, passando através do servidor de redirecionamento. Deve-se elaborar uma estruturação que se enquadre nas especificações CORBA, isto é, permita

que mesmo separados pela rede, os clientes e o servidor possam interagir de forma satisfatória e sem o aumento da complexidade dessas classes. O uso da Linguagem de Definição de Interfaces (IDL) do CORBA possibilita determinar o nível mais alto dessa estruturação, ou seja, as interfaces por onde se comunicarão o cliente e o servidor. Nota-se que não é mencionado o serviço, nesse nível da arquitetura. Sua participação na comunicação não depende do CORBA, sendo tratado diretamente pelo servidor, que faz uso dos processos de segurança.

Através da definição em IDL, o pré-processador IDL gera automaticamente uma estrutura de classes que se torna responsável pela manipulação e adequação dos dados a serem transportados. Entre os elementos gerados pelo pré-processador IDL para essa estrutura está a classe *LarrayHolder*, que é correspondente a um *array* de *bytes*, para parâmetros passados por referência (discutido na Seção 3.5.4). Como essa estrutura é automaticamente gerada, é mais importante salientar a criação das classes responsáveis pela comunicação entre o cliente e o servidor. Essa interface de comunicação é constituída por classes relacionadas, os *stubs* e os *skeletons*. Os *stubs* referem-se ao cliente, enquanto os *skeletons* referem-se ao servidor; eles definem os métodos de comunicação com o ORB e entre si, de modo que só resta ao programador implementar as operações específicas do sistema, uma vez que é transparente a comunicação entre as duas partes, através do CORBA. (Figura 19).

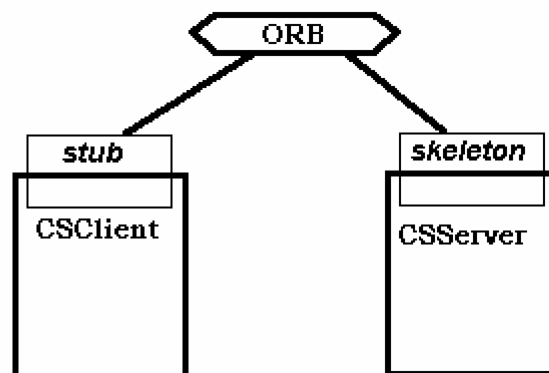


Figura 19 _ Ligação Cliente/Servidor Através do ORB

Seguindo, então, a estrutura definida para este trabalho, uma aplicação que deseja se conectar a um serviço específico, através de um servidor de redirecionamento com segurança de dados irá instanciar um objeto da classe cliente *CSClient*, que atende a requisições semelhantes às de uma classe *socket* convencional.

Dessa forma, para modificar uma aplicação já existente, é necessário somente sobrescrever (*override*) a classe *Socket* convencional que está em uso. A classe *CSClient* cria (ou contacta) o ORB da máquina local, e através das interfaces definidas no *stub*, conecta-se ao servidor específico.

A implementação do CORBA escolhida para essa arquitetura inclui o chamado *Factory Server*. Esse servidor tem como finalidade criar servidores de redirecionamento (classe *CSServer_impl*, construída com os *skeletons* do IDL) sob demanda, um para cada cliente. Essa facilidade é requerida devido à característica não convencional de manter uma conexão *socket* para o serviço requisitado ativa no lado do servidor, específica para cada cliente. Um servidor *Factory* instancia objetos da classe servidora à medida em que são requisitados. A eliminação dos objetos instanciados ocorre quando sua conexão ao serviço é finalizada, através de um método específico, caracterizando o fim da sessão de comunicação, e que desconecta o objeto do ORB, tornando-o passível da *Garbage Collection* (coleta de lixo) automática do Java. (Figura 20).

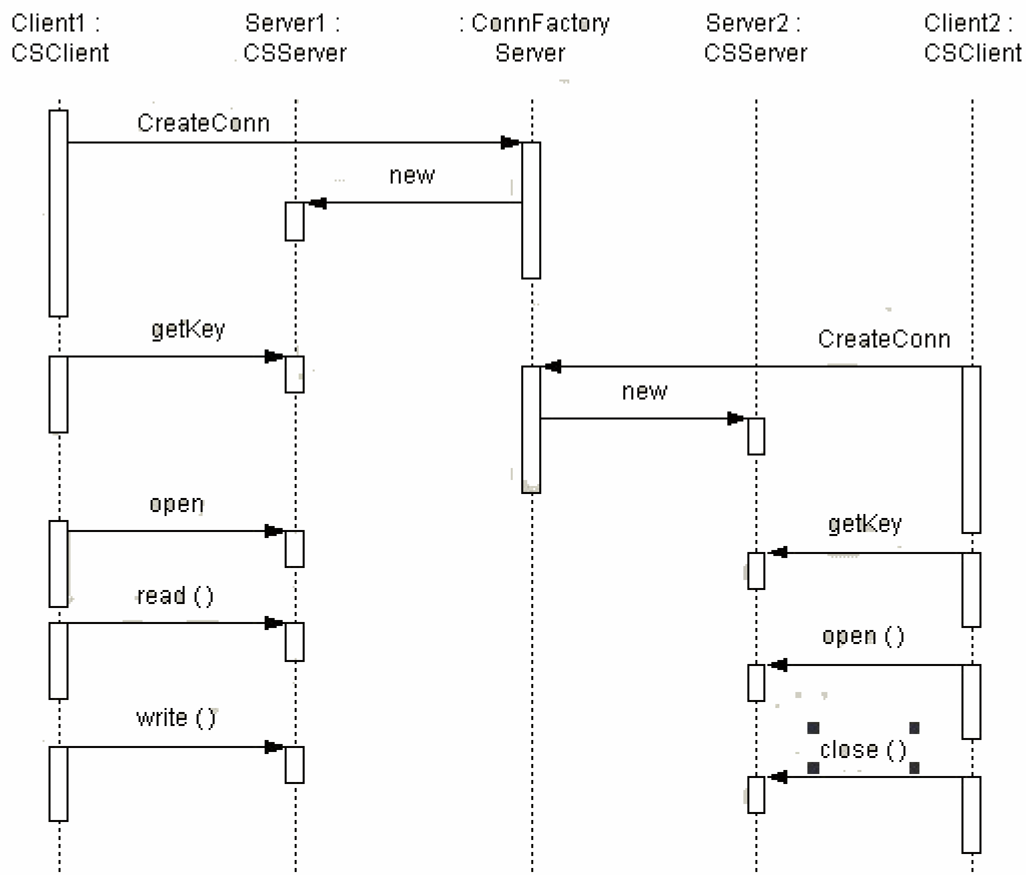


Figura 20 _ Designação de Servidores pelo *Factory Server*

4.3 Adicionando Privacidade à Mensagem

Para o tratamento dos dados trafegados entre o cliente e o servidor é necessária também uma estrutura específica para essa finalidade. Como o processo envolve várias etapas (geração e validação de *tickets*, serialização, criptografia e assinatura digital), executadas em uma seqüência determinada, as classes geradas para a execução de tarefas individuais foram relacionadas ao redor de uma classe coordenadora, chamada de *Packer*, que corresponde ao centro de transformação dos dados visível ao ambiente externo. A existência dessa única interface com o conjunto de classes responsáveis pelo processamento dos dados, leva à uma independência maior sobre as funções efetuadas, tornando possível adaptar essa solução como um componente reutilizável em sistema diversos.

Cada uma das etapas acima, a serem executadas, têm características diferentes, motivo que leva cada uma a ter um comportamento e uma existência diversa das demais.

4.3.1 Tickets

As tarefas envolvendo a geração e a autenticação de *tickets*, implementadas na arquitetura através da classe *TicketUtil*, tornam obrigatória a preservação do objeto responsável pelo processamento por todo o tempo de existência da conexão. Isso é requerido por causa da verificação de *tickets* necessitar a preservação dos dados anteriormente processados, de forma a detectar possíveis falhas de segurança. Os *tickets*, por sua vez, são utilizados um a cada nova transmissão, o que os torna muito voláteis.

A definição sobre a construção dos *tickets* levou a certas decisões de projeto. De um lado, existem todos os conceitos de especialização dos objetos, e as vantagens do uso de classes, essencialmente modulares, como forma de reduzir os custos de uma possível modificação do sistema. Por outro lado, existe o conceito de reutilização de estruturas com funcionalidades ou objetivos semelhantes aos requeridos. Nessa escolha, optou-se por reutilizar uma classe de objetos já definida pela própria linguagem Java, a classe *Vector*, que permite o armazenamento de objetos de tipos diferentes em uma mesma lista. Uma solução alternativa, que foi descartada devido à

facilidade de reaproveitamento de classes, envolvia a criação de uma classe específica para a armazenagem dos dados. Apesar desta decisão, não há compromentimento de uma futura extensão do sistema, pois a operação de armazenagem dos *tickets* ocorre sem dependência de outros sistemas, permitindo uma rápida alteração das classes. Na Figura 21 são visualizados a estruturação da classe *TicketUtil*, responsável pela geração de novos *tickets* e a validação dos *tickets* recebidos, e sua relação com a classe reutilizada do Java.

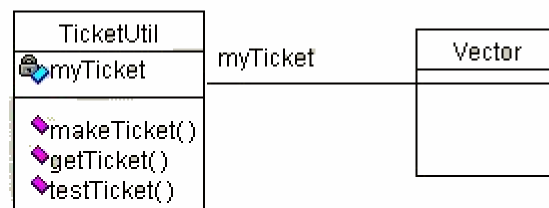


Figura 21 _ Classe *TicketUtil*

4.3.2 Serialização

Para a adequação do pacote de dados, que contêm os *tickets* e a mensagem, à uma representação passível de cifragem, foi determinada a classe *Serializador*. Sua tarefa é efetuar as operações de serialização de objetos, especificada pela linguagem Java como uma maneira de descrever um objeto em sua estrutura e conteúdo, transformando-o em dados primitivos, e possibilitar sua remontagem. É a essa estrutura de dados que são aplicados os processos criptográficos. Sendo utilizada apenas como centro de transformação de dados, um objeto gerado persiste até a finalização da conexão.

4.3.3 Criptografia

Os processos criptográficos mencionados anteriormente são essenciais à atribuição de segurança aos dados trafegados. A implementação do método criptográfico escolhido é realizada através da utilização das classes descritas no

pacote *Cryptonite* (Anexo IV). As classes ali descritas são responsáveis pela construção de chaves criptográficas e pelas operações de cifragem e decifragem, sendo utilizadas no desenvolvimento do sistema as classes *Key*, que armazena uma única chave criptográfica, e *keyPair*, que armazena o par de chaves pública e privada.

A fim de encapsular os procedimentos adequados à cifragem dos dados, e para manter a persistência das chaves criptográficas utilizadas tanto para a criptografia quanto para a assinatura digital dos dados, a classe *Encryptool* foi criada para coordenar essa etapa final do processamento dos dados. Sua tarefa, apesar de ser uma das mais importantes do sistema, é simples de ser executada (considerando a utilização das classes criptográfica do pacote *Cryptonite*). Desse modo, Sua atuação restringe-se ao processamento dos dados, finalizando a transformação realizada através das classes do sistema.

A Figura 22 mostra o relacionamento entre as classes responsáveis pelo processamento dos dados. Essas classes estão presentes tanto no cliente quanto no servidor, pois são responsáveis também pelo processo de extração da mensagem cifrada.

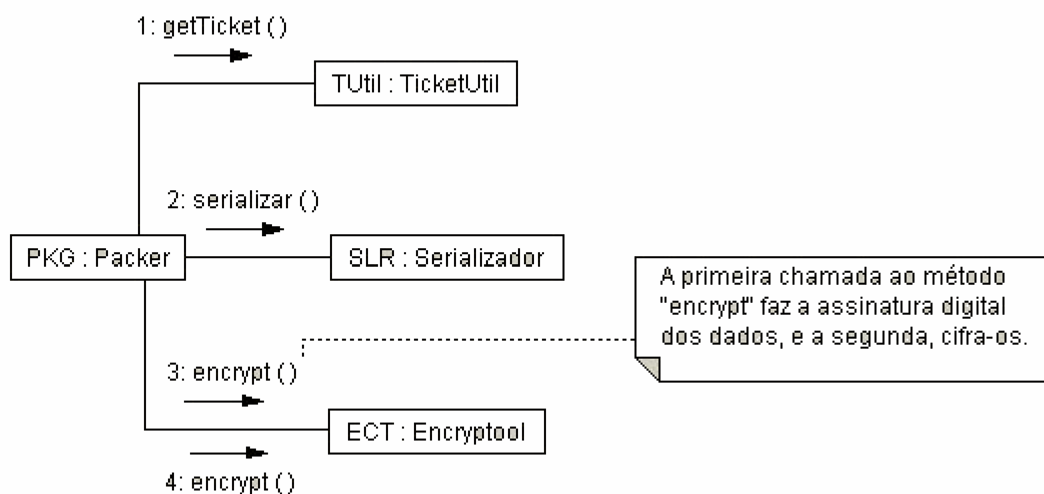


Figura 22 _ As Classes Criptográficas do Sistema Cifrando uma Mensagem

4.3.4 A classe *Packer*, coordenadora da transformação

A relação das classes do cliente e do servidor com o processamento dos dados dá-se através da classe *Packer*, e a invocação dos métodos referentes à classe *Packer* são realizados pelos métodos do cliente e do servidor (Figura 23), de acordo com suas funções, semelhantes (ainda que não tenha sido possível herdá-las diretamente) às da classe *Socket* do Java e suas complementares.

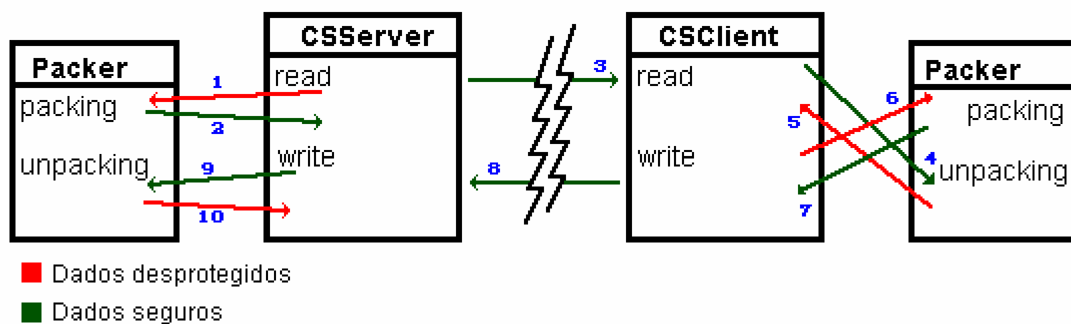


Figura 23 _ Processos de Cifragem sobre os Dados Enviados

Além disso, o servidor ainda mantém uma conexão através de *sockets* tradicionais com o *host* de origem do serviço, atuando também no redirecionamento das mensagens entre o cliente e esse serviço (Figura 24).

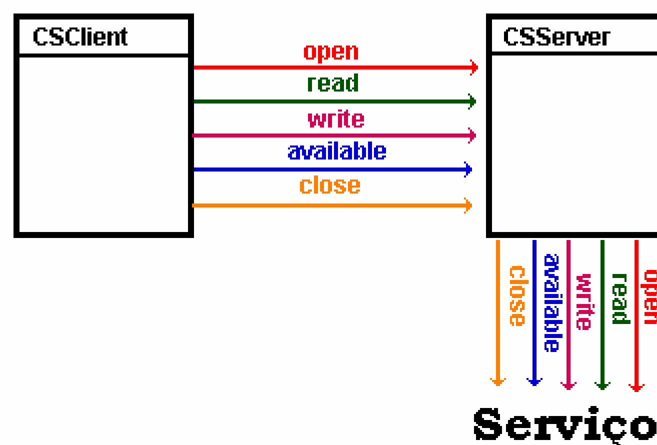


Figura 24 _ CSServer Atuando no Redirecionamento

A forma como foi construída a estrutura das classes tornou independentes os sistemas de redirecionamento (nessa implementação é usado o CORBA) e o sistema de criptografia e autenticação (representados pelas classes comandadas por *Packer*). Essa modularidade pode ser útil tanto para realizar testes de eficiência sobre métodos criptográficos e de autenticação, quanto para a utilização da solução de segurança implementada nesse trabalho com um componente "plugável" a diversos sistemas de comunicação.

4.4 Clientes e Servidores

Como os clientes e os servidores utilizam classes diferentes, de acordo com sua participação na comunicação CORBA, as estruturas desenvolvidas também são ligeiramente diferentes para cada uma das partes. Enquanto o servidor (Figura 26) é criado em função de um ORB já existente na sua máquina (devido ao *Factory Server*), um cliente tem que buscar e manter as referências desse ORB e desse servidor, para realizar a conexão (Figura 25).

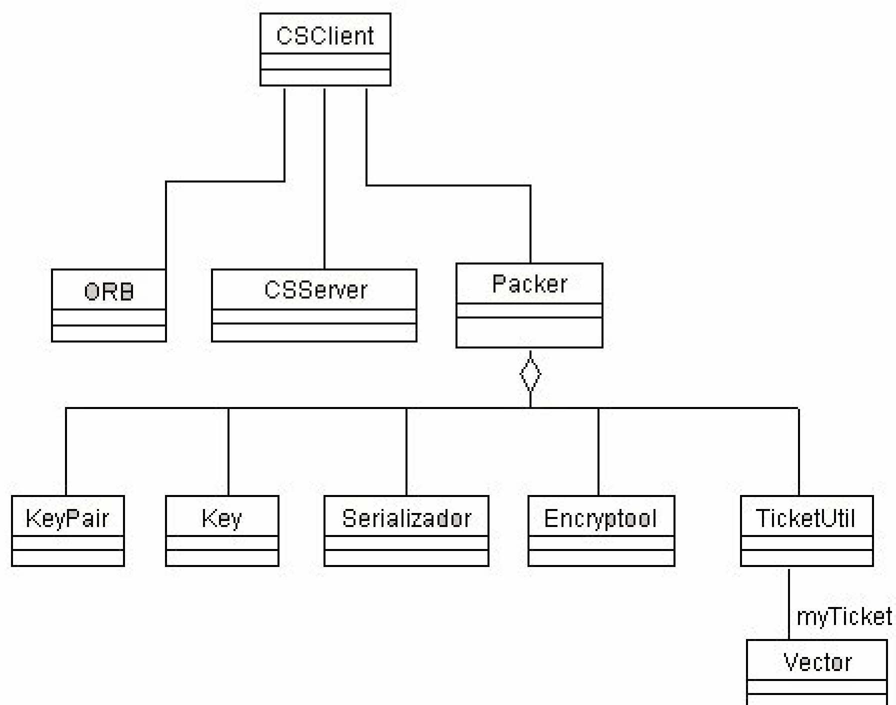
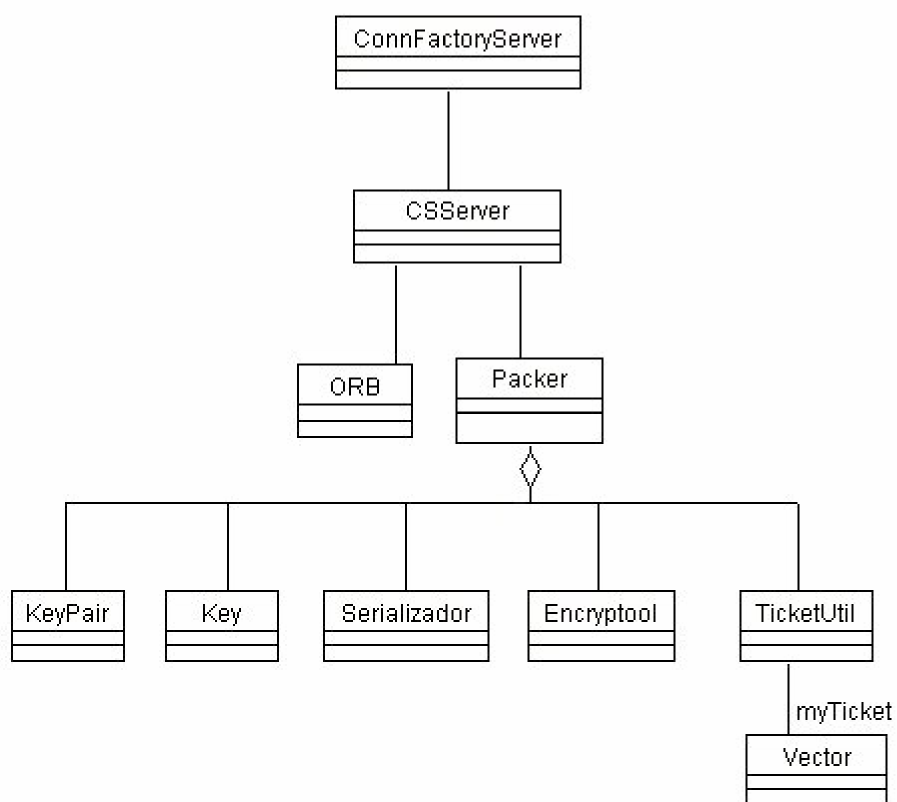


Figura 25 _ Hierarquia de Classes do Cliente

A presença das classes de criptografia em ambos lados da comunicação é essencial, já que as operações de cifragem e autenticação demandam um conhecimento equivalente das duas partes envolvidas.

Como as classes necessárias ao cliente são diferentes do servidor, é interessante reuni-las separadamente, para minimizar o tamanho do programa transportado pela rede para o *browser* (a ser discutido no Capítulo 5).



5 APLICAÇÕES DESENVOLVIDAS

5.1 A Escolha das Aplicações

Como o sistema desenvolvido não constitui por si só uma aplicação, e sim um conjunto de classes que visa estender as funcionalidades de comunicação utilizadas em diversas aplicações, há a necessidade de testar sua eficácia através da sua implementação em um sistema auxiliar, que funcione como base para os testes.

A possibilidade de desenvolver uma aplicação específica para esse fim foi descartada, pois além de desviar os esforços sobre o desenvolvimento das classes de comunicação, é muito passível de se moldar aos hábitos do programador, e não representar as condições reais de uso.

A primeira aplicação escolhida para ser estendida foi um *applet* que emula um terminal TN3270 em Java. Essa aplicação, desenvolvida na Universidade de Genebra[7], na Suíça, em 1995, ofereceu um ótimo ambiente de trabalho, pois é um sistema completo, que trata de todos eventos relacionados à emulação do terminal, e possibilita que o programador concentre-se somente na extensão da comunicação. Essa tarefa também é facilitada devido à estruturação das classes da aplicação, que reúne as operações de comunicação em uma única classe, *Tn3270tcp*, permitindo uma fácil adequação.

Outra característica da aplicação *TN3270,* que foi estudada, refere-se à possibilidade de criar formas diversas de apresentação, além do sistema em modo texto tradicional, podendo adicionar características de interatividade que o Java propicia para tornar a utilização mais agradável esteticamente e ao uso. Especificamente para a aplicação de demonstração[8] disponibilizada pelos autores, foi desenvolvido o acesso ao sistema gerenciador da biblioteca da universidade, apresentado através de duas formas de interação: o primeiro, usando a interface tradicional dos terminais *TN3270* (modo texto), e o segundo, através de um ambiente gráfico, que contém o histórico do acesso visualizado através de uma árvore de diretórios. Entretanto, esse mapeamento gráfico é muito dependente da aplicação

[7] *http://www.unige.ch/hotjava*
[8] *http://www.unige.ch/hotjava/HotSibil.html*

executada pelo *TN3270*, e não há uma documentação específica, apenas a experiência dos criadores descrita nas classes implementadas.

A outra aplicação de testes, um terminal Java para um acesso remoto através de *telnet*, intitulada *The Java(tm) Telnet Applet*[9], desenvolvida por Matthias L. Jugel e Marcus Meißner e disponibilizada através da licença de uso GPL, foi escolhida bem mais ao fim da implementação, e serviu principalmente para testar a funcionalidade das classes desenvolvidas, utilizando-as em aplicações distintas, que compartilham o mesmo servidor de redirecionamento. Esta aplicação de *telnet* contém diversos módulos e tipos de terminais, adaptando-os a usos específicos, como serviços de IRC ou de MUD, e a documentação das classes é bem detalhada quanto a esses aspectos. A adaptação às classes desenvolvidas também foi muito bem sucedida, embora algumas características dos terminais *telnet* ainda tornem a aplicação levemente instável, devido principalmente à performance dos processos de cifragem, e que será discutido na Seção 5.3.

Outro fato interessante, relativo a ambas aplicações, é que para fugir às restrições de acesso à rede impostas pelos *browsers* ao Java, as aplicações possuem suas próprias soluções para redirecionamento, normalmente através de um *daemon* que fica rodando na máquina servidora, mas que não conta com o sistema de cifragem de dados que caracteriza a arquitetura desenvolvida.

5.2 A Aplicação TN3270 Java

A escolha de uma aplicação de *TN3270* para testar a arquitetura desenvolvida deve-se principalmente à um questionamento feito pelos programadores e pelos administradores de rede do Centro de Processamento de Dados da Universidade Federal de Santa Maria: como atualizar os serviços oferecidos pela universidade através de terminais *TN3270*, adaptando-os às novas exigências dos sistemas cliente-servidor com custo reduzido e rapidez de implementação, e também como distribuir essas aplicações de forma segura, possivelmente através de um *firewall* instalado para a rede da universidade.

---

[9] *http://www.first.gmd.de/persons/leo/java/Telnet/*

Utilizando as classes de comunicação desenvolvidas, é possível distribuir as aplicações antigas, mas com uma interface gráfica amigável, com privacidade sobre os dados e sem a necessidade de contato direto da Internet com o computador onde rodam os sistemas.

Das aplicações testadas, esta foi a que melhor se adaptou às classes utilizadas para garantir a segurança dos dados trafegados. Devido à característica dos terminais *TN3270* de enviar "páginas" de dados, ao invés de caracteres individuais, foi possível otimizar o *overhead* da cifragem dos dados, e reduzir a freqüência das requisições de cifragem e decifragem dos dados, que freqüentemente causa problemas no *applet* de *telnet*.

Na Figura 27, pode ser visualizada a estrutura em que se organiza a aplicação TN3270. É nessa estrutura que serão encaixadas as classes de comunicação, substituindo as funções dos *sockets* convencionais.

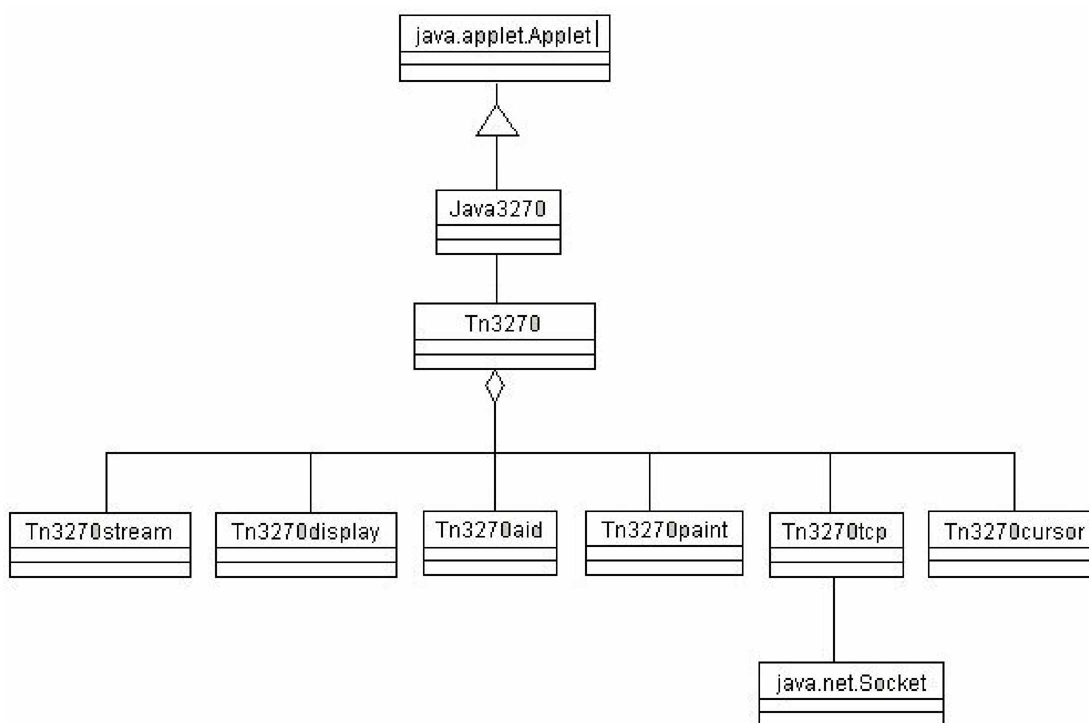


Figura 27 _ Estrutura do *Applet Tn3270*

A integração das classes de criptografia à estrutura do *applet TN3270* dá-se através da substituição da referência à classe *java.net.Socket* pela classe *CsocketS.CSClient*. Como a classe *CSClient* tem interfaces (métodos) semelhantes em

nome e em função às da classe *Socket*, não é necessário submeter o restante das classes do TN3270 à alguma modificação adicional. Na Figura 28 vê-se como essa substituição não altera o funcionamento das chamadas.

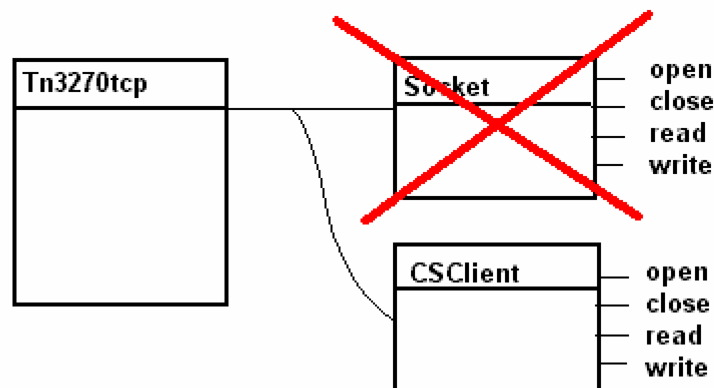


Figura 28 _ Substituição da Referência à Classe *Socket*

Para uma melhor performance no *download* das classes do *applet*, incluindo as classes do ORB requisitadas pelo cliente, o *applet* teve suas classes reunidas em um arquivo único, denominado *TN3270Client.jar*. Esse formato de arquivo é definido pela linguagem Java, e apresenta-se de forma semelhante aos arquivos *tar* do *Unix*, tendo como principais objetivos reduzir o número de arquivos separados para *download* (a transmissão de um bloco grande é mais vantajosa do que de vários arquivos), e realizar a compressão das classes onde for possível, para melhorar a velocidade da transmissão.

Infelizmente, esse arquivo não é único para todas as instalações de um servidor. Como a referência CORBA ao servidor é feita de forma estática, através da obtenção da referência IOR disponibilizada pelo servidor, há a necessidade de configurar o cliente para buscar essa referência no *site* do servidor. Foi elaborado um *script* Unix para atualizar automaticamente esses dados, e montar um novo arquivo *jar* personalizado para acessar o servidor designado na instalação. Não foi desenvolvido um similar para o Windows, mas as modificações são as mesmas.

Apresentando como resultado final está o *applet* modificado, que é capaz de se comunicar com um servidor *VM/ESA*[10] (que normalmente utiliza terminais *TN3270*) e acessar seus serviços remotamente (Figura 29).

[10] Sistema Operacional da IBM Corp.

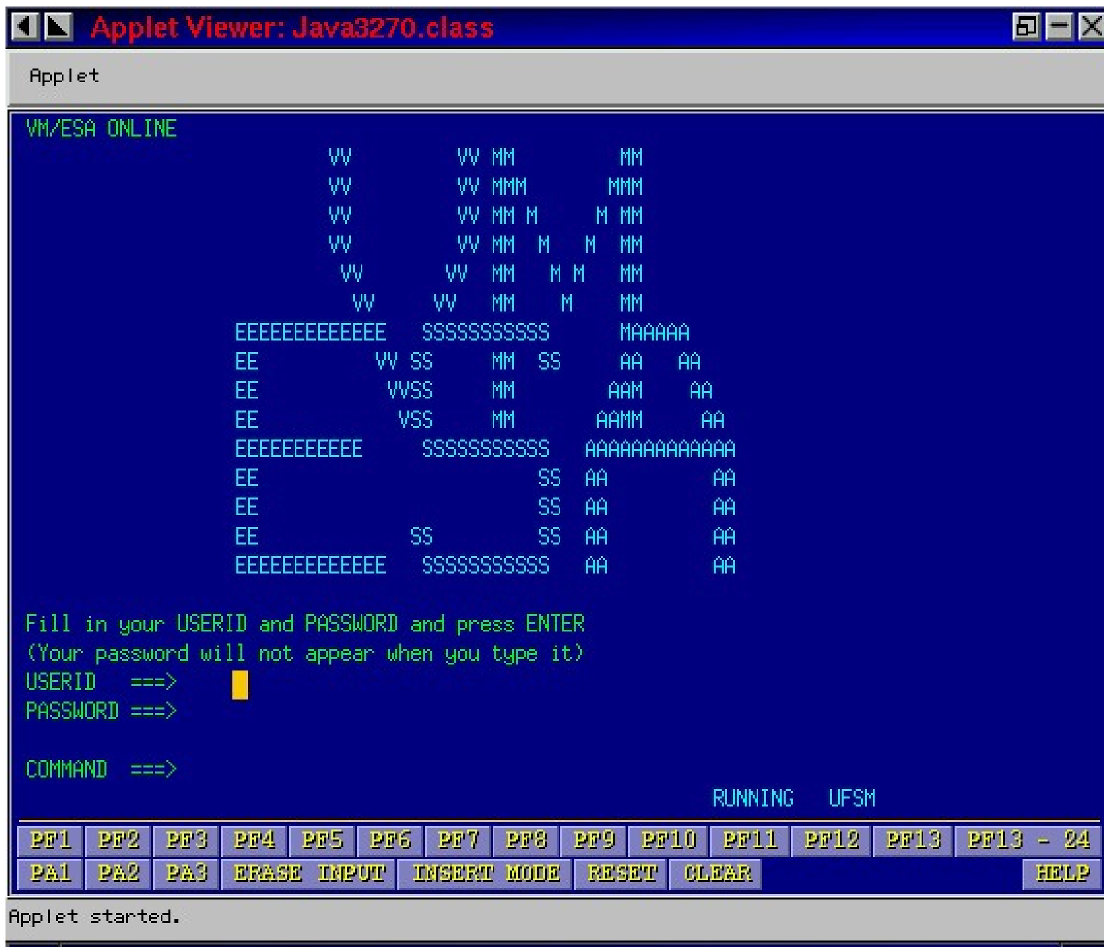


Figura 29 _ Tela de Abertura de uma Sessão TN3270

5.3 A aplicação Telnet

Ao contrário da aplicação de *TN3270*, o *applet* de *telnet* somente foi testado quando as classes do sistema de segurança já estavam na fase final, e haviam sido testadas no applet de *TN3270*. A experiência com o *applet* de *telnet* foi motivada inicialmente apenas por curiosidade, pois apesar de ter respeitado o formato do *socket* padrão, não era esperado o sucesso na implantação das classes desenvolvidas em outras implementações, ao menos sem modificações muito específicas.

A única modificação realizada foi a adição de mais um método dos *socket*s, *available*, mudança essa que não alterou o funcionamento da aplicação *TN3270* pois o novo método é utilizado somente pelo *telnet*.

Devido à essa experiência não ter sido programada para o desenvolvimento do trabalho, não foi possível estudar mais a fundo o funcionamento do *applet*. Entretanto, os criadores geraram uma ótima documentação. A página principal da documentação é apresentada no Anexo III.

Assim como o *applet* de *TN3270*, o *telnet* possui uma estrutura modular, que torna fácil a manutenção da classe relativa à comunicação. Sua estrutura total, no entanto, é muito mais complexa que a do *TN3270*, pois possui diversas opções de módulos e terminais "plugáveis" de acordo com a funcionalidade requerida. Como o tempo de análise dessas classes foi muito reduzido, optou-se por estudar apenas as classes relacionadas diretamente à comunicação, conforme demonstra a Figura 30.

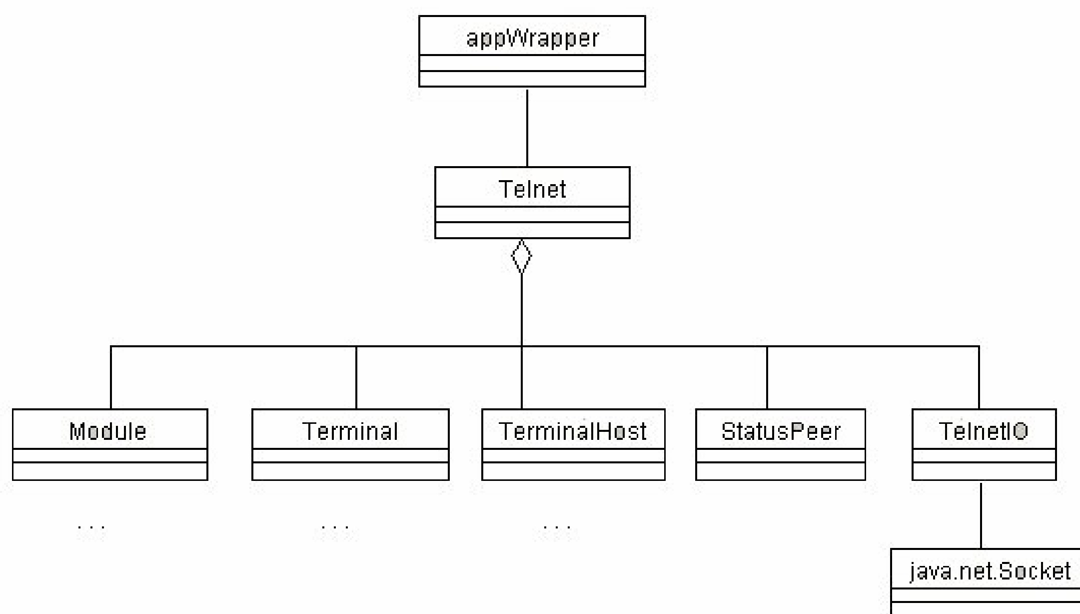


Figura 30 _ Parte da Estrutura do *Applet Telnet*

Observando esse esquema, percebe-se que assim como no *applet TN3270*, a classe *Socket* pode ser substituída sem perdas pela classe *CSClient*, a interface do cliente dentro do sistema de segurança e redirecionamento.

Entretanto essa aplicação não se adequa totalmente ao sistema de autenticação de *tickets* empregado. Considerando que um *telnet* envia e recebe os dados caractere a caractere, uma entrada de dados muito rápida, ocasiona a perda da sua seqüência de *tickets*, quando são geradas novas mensagens antes do retorno da mensagem do

primeiro *ticket*, obrigando a aplicação a fechar a conexão devido aos *tickets* antigos não serem mais reconhecidos (Figura 31).

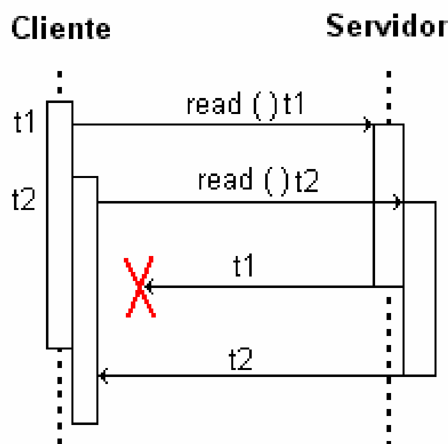


Figura 31 _ Falha na Identificação dos *Tickets*

Essa perda de contato ocorre quando o usuário digita com velocidade superior à velocidade de resposta do sistema.. Num *telnet*, cada caractere digitado tem que ser processado pelo servidor para retornar e ser mostrado na tela. Quando são enviadas duas mensagens (e portanto, gerados dois *tickets* diferentes), antes que a visualização da primeira retorne, o sistema perde o contato com o *ticket* anterior, pois não há a manutenção de uma lista dos *tickets* anteriores, para comparação.

Uma possível solução, a ser estudada e implementada em um trabalho futuro, a troca do sistema de envio, mantendo um *buffer* de envio, cujos dados são transmitidos de forma agrupada, exigindo apenas a geração de um *ticket* por mensagem e ainda diminuindo o *overhead* causado pelo processamento criptográfico sobre cada mensagem. A determinação de um tempo limite para a transmissão do *buffer* garante a confiabilidade e o tempo de resposta que esse tipo de conexão necessita.

Apesar dessa limitação, a aplicação de *telnet* também é plenamente funcional, desde que seja respeitado um limite de velocidade para a digitação dos dados.

A Figura 32 mostra o *applet* de *telnet*, já adaptado às classes desta arquitetura, em utilização para *login remoto*.

Assim como o *applet* de *TN3270*, as classes do *applet* de *Telnet* foram reunidos em um arquivo *jar*, e um *script* desenvolvido para adaptá-lo à localização de origem do servidor.

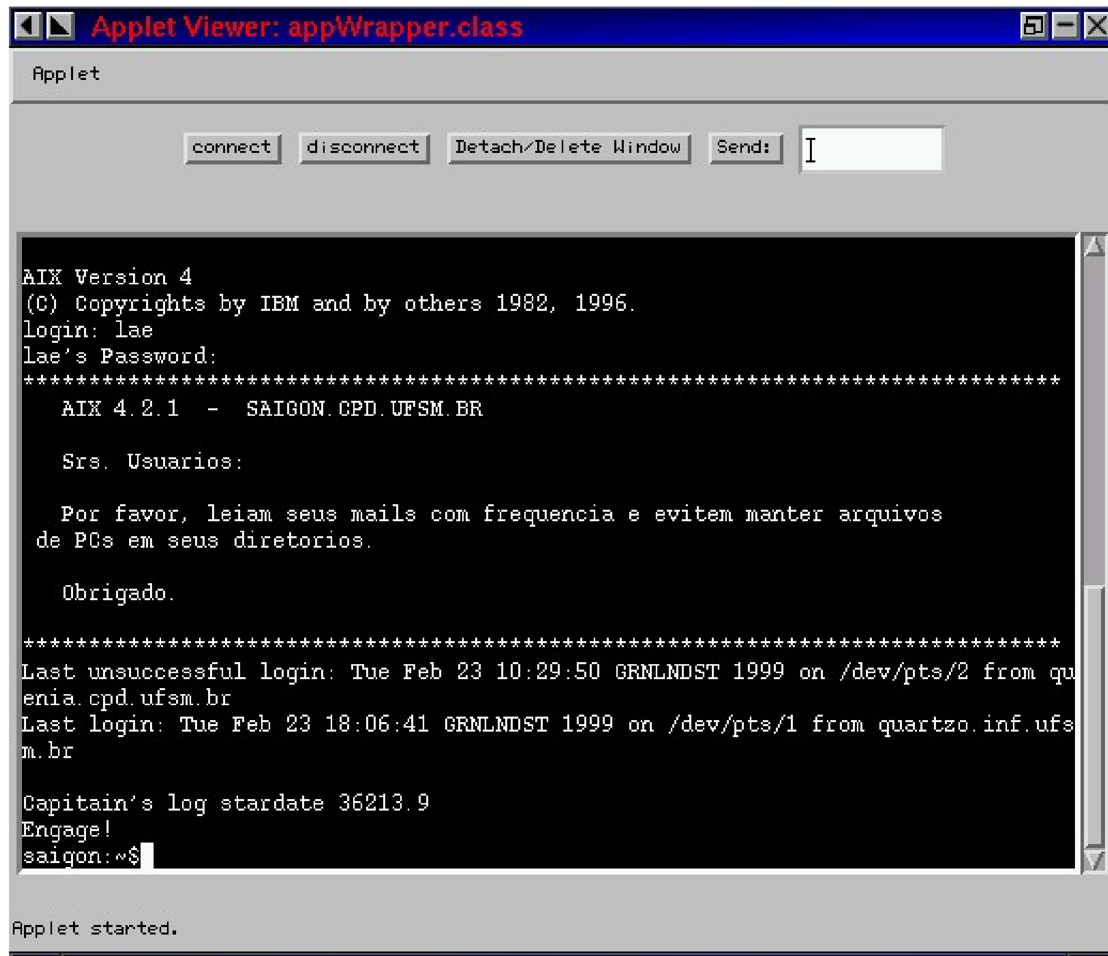


Figura 32 _ Sessão de *telnet* Aberta com o *Applet*

5.4 Observações Finais

Duas observações devem ser feitas sobre as aplicações estendidas, por serem comuns às duas: seleção do serviço de destino, e o comportamento dos *browsers* mais comuns.

A primeira observação refere-se à escolha do endereço que se deve contactar para prover os serviços. Como ambas implementações estabeleceram esse tipo de

entrada de dados através de parâmetros do próprio código *html* que chama o *applet*, esse arquivo também deve ser personalizado para cada serviço diferenciado que se desejar. A segunda observação visa mostrar que apesar de ser conhecida por sua independência de plataforma e sistema, a linguagem Java ainda não está similarmente suportada entre os principais *browsers* que a suportam.

Testes feitos utilizando o *Netscape Navigator 4.0x e 4.5* foram bem sucedidos tanto no sistema operacional *Linux* quanto nos sistemas *Windows 95 e NT*. A dificuldade em utilizar esse *browser* reside no fato dele manter um *ORB* interno, mas a solução para isso é facilmente encontrada, sem custos ao sistema (conforme descrito na Seção 4.1).

O *Internet Explorer 4*, testado nos sistemas *Windows*, não apresenta um ORB interno, mas no entanto apresenta duas falhas na execução das aplicações, derivadas da implementação da Máquina Virtual Java (JVM). A primeira refere-se à exibição correta das letras, pois o *Explorer* freqüentemente confunde as letras a serem exibidas, tornando ilegível os dados apresentados. A Segunda, refere-se ao tratamento do caso sensitivo. Embora os terminais TN3270 sejam insensíveis a esse contexto, o *Explorer* não consegue transmitir corretamente os comandos escritos em letras minúsculas, que retornam mensagens de erro.

Por último, o *appletviewer*, *browser* de *applets* que vem na distribuição do *Java Development Kit* 1.1.5 (JDK). Mesmo sendo um sistema desenvolvido pela própria *Sun Microsystems*, autora da linguagem Java, o *appletviewer* para *Windows* não reconhece os eventos de teclado (pressionar uma tecla, por exemplo) descritos na versão 1.0.x da linguagem, inclusive para outras aplicações que utilizem esses eventos. Isso torna incorreta a afirmação de que o Java suporta plenamente os métodos ultrapassados das versões anteriores. No *appletviewer* para o *Linux*, portado por grupos não diretamente relacionados com a *Sun Microsystems*, esses eventos são reconhecidos.

6 CONCLUSÃO

As classes *CsocketS* desenvolvidas com o propósito de criar uma conexão de rede semelhante aos *sockets* adicionada de criptografia e autenticação, representam uma solução diferente e de custo reduzido para disponibilizar serviços tradicionais de rede, com confiabilidade e sigilo.

O modo como as classes foram projetadas, de forma a realizarem tarefas similarmente aos *sockets* possibilita a adequação de sistemas já existentes a um ambiente de comunicação seguro, mas sem a necessidade de alterar a estrutura dessas aplicações, nem requisitar o aprendizado de uma nova metodologia de acesso.

Da forma como foi implementado, o sistema tem como grande mérito a maximização da segurança dos dados transmitidos, em aplicações onde a interação com o usuário costuma limitar a confiabilidade desse sigilo. Essa segurança é obtida não através de um único e oneroso método de cifragem dos dados, mas através da utilização integrada de dois sistemas de autenticação, os *tickets* e a assinatura digital, e um método de criptografia de custo mais reduzido. Essa aplicação simultânea de diversos métodos com custo menor possibilita uma segurança total aos dados durante sua transmissão, por tempo suficiente para realizar todas operações necessárias em sigilo.

Isso não quer dizer que não possam ser usados métodos criptográficos mais robustos, ou com maior complexidade das chaves, mas o custo representado pelo tempo de processamento desses dados cifrados, com os recursos computacionais de hoje, inviabiliza uma resposta rápida ao usuário, o que modifica a proposta das aplicações. Aplicações que dependem menos do usuário podem considerar esse incremento da segurança, ao custo da velocidade, ou caso necessário, o uso de *hardware* específico, como os processadores com suporte nativo à linguagem Java.

Em casos onde é necessário uma maior eficiência do processamento dos dados, pode-se utilizar compiladores *Just-in-Time* (JIT) pode acelerar o processo. Nesses compiladores, o programa Java é compilado para a arquitetura específica antes de ser executado, garantindo uma performance superior ao código interpretado pela Máquina Virtual Java. Essa solução também é útil para o principal ponto de estrangulamento do sistema, que ocorre no servidor quando há muitas requisições de conexão. O uso de

um compilador JIT ou sistemas similares (existem sistemas que convertem o código Java para a linguagem C) traduz-se em uma eficiência superior para o servidor.

Caso o sistema de cifragem empregado ainda represente uma carga onerosa ao sistema, pode-se projetar modificações no sistema de cifragem. Ao contrário do método utilizado atualmente, através de criptografia RSA pura, pode-se implementar as chaves RSA apenas para a troca das chaves secretas DES, cujo processamento é mais veloz, a exemplo do sistema *ssh*. Cabe a um próximo trabalho verificar a viabilidade e a eficácia de tais otimizações, e o teste de diversas soluções em conjunto.

Ainda assim, o sistema desenvolvido é eficaz, e enquanto a aplicação de *telnet* apresenta certas limitações, a aplicação de *TN3270* adapta-se muito bem ao modelo desenvolvido. Ambas aplicações, inicialmente com o acesso à rede restrito pelos *browsers* compatíveis com Java, agora superam essas limitações através do servidor, que age como redirecionador de conexões, estando totalmente adequadas à implementação em uma rede protegida por *firewalls*. E é bem provável que futuras aplicações desenvolvidas diretamente sobre as classes de comunicação segura aproveitem mais eficientemente os recursos e características de uma comunicação não tradicional.

Uma característica notável do sistema desenvolvido foi obtida a partir da associação das classes que simulam um *socket* ao modelo de distribuição de objetos CORBA. Um servidor das classes CsocketS aceita indistintamente requisições de ambas aplicações, *TN3270* e *telnet*, integrando o acesso aos serviços em um mesmo servidor. Mais do que somente dividirem a mesma aplicação servidora, cada cliente recebe um processo-servidor individual, que irá gerenciar sua conexão ao serviço requisitado com dedicação exclusiva.

O sistema completo, incluindo o código fonte das classes, a documentação e as aplicações adaptadas, será disponibilizado para *download* através do endereço *http://www.inf.ufsm.br/~lae.*

6.1 Trabalhos Futuros

O sistema gerado possibilita diversas extensões, gerando um campo fértil para o desenvolvimento de outros trabalhos.

Os trabalhos que venham a estender as opções do sistema poderiam empenhar-se em minimizar o custo do processo criptográfico, testando formas diversas de conseguir prover segurança aos dados sem onerar tanto o sistema com o processamento dos dados cifrados. Pode-se modelar um novo sistema de autenticação, que evite o problema da perda de contato com os *tickets* anteriores, mas que também não restrinja a liberdade de transmissão das aplicações.

O sistema pode ainda ser utilizado para oferecer serviços não somente de redirecionamento de conexões *socket*. Um servidor que concentre as requisições ODBC/JDBC, e as distribua aos sistemas gerenciadores de banco de dados de acordo com suas cargas de processamento ou conteúdo, por exemplo, ou um leitor de e-mail através do protocolo POP-3 que transmita os dados do usuário e receba os mails em segurança (normalmente, todos os dados são transmitidos abertamente, inclusive a senha).

Um uso mais específico refere-se ao emprego das classes de comunicação para prover um sistema de armazenamento de dados cifrados, através da serialização. Por exemplo, supondo que o processo da declaração do Imposto de Renda ainda não implemente um sistema de segurança, os dados poderiam ser transmitidas de forma segura, com o uso das classes desenvolvidas, e armazenadas em modo cifrado até que sua utilização fosse realmente necessária. A geração das chaves do contribuinte poderia ser relacionada aos seus dados, evitando falsificações, ou ao menos atribuindo a característica de não refutação.

7 BIBLIOGRAFIA

[BIR 95] Birman, Kenneth P. *Building Secure and Reliable Network Applications* Department of Computer Science - Cornell University. 1995.

[COR 97] Cornelius, Barry. *Developing Distributed Systems*. Apostila, *Barry.Cornelius@durham.ac.uk*. 1997.

[COR 98] Cornelius, Barry. *Using CORBA and JDBC to produce Three Tier Systems*. Apostila *Barry.Cornelius@durham.ac.uk*. 1998.

[JDC 98] *Sun Mycrosystems*. *Java Developers Connection Tech Tips*. Vol. 1 Número 7, *JDCTechTips@javasoft.com*, 1998.

[JDK 97] *Sun Microsystems*. *Java Development Kit 1.1.5 API Documentation*. *http://www.javasoft.com* . 1998.

[KRI 97] Sundararaman, Krishnan. *Client-Server in Java (Parts I, II e III).* Apostila, *chris@OLABS.COM,* 1997.

[MOR 98] Morin, Richard. *DES Vérités*, Revista SunExpert vol.9 número 10, pág. 32-35, 1998.

[OMG 97] *Open Management Group. IDL Java Language Mapping*. Definição de padrão (97-03-01). 1997.

[OOC 98] *Oriented Object Concepts. ORBacus 3.1.1 documentation*. Apostila, *http://www.ooc.com*. 1998.

[ORF 97] Orfali, Robert e Harkey, Dan. *Client/Server programming with Java and CORBA*. John Wiley & Sons Inc., 1997.

[PIN 97] Pinheiro, Manuele K. e Augustin, Iara. *Simulando um Compilador na Web*, Trabalho de Graduação – Bacharelado em Informática, Universidade Federal de Santa Maria. *http://www.inf.ufsm.br/~manuele*, 1997.

[RSA 98] RSA Laboratories. *RSA Laboratories Frequently Asked Questions About Today's Cryptography v.4.0.*, *http://www.rsa.com/rsalabs/faq/*, 1998.

[SUN 97] *Sun Microsystems*. *CORBA for Java Programmers*. Apostila *http://www.javasoft.com*. 1997.

[TAN 95] Tanenbaum, Andrew S. *Sistemas Operacionais Modernos*. Editora Prentice-Hall do Brasil, 1995.

[VOG 97a] Vogel, Andreas e Duddy, Keith. *Java programming with CORBA*. Wiley Computer Publishing, 1997.

[VOG 97b] Vogel, Andreas. *Client/Server Development with Java and CORBA*. Apostila "Objetos Distribuídos '97 - Curitiba - PR". 1997.

[WEB 94] Weber, Raul. *Polígrafo sobre Criptografia*, Universidade Federal do Rio Grande do Sul, 1994.

[WEB 97] Weber, Raul. *Criptografia Contemporânea*. Universidade Federal do Rio Grande do Sul, 1997.

[WEL 96] Wells, David. *Internet Tools Survey – Authentication*. *http://www.objs.com/survey/authent.htm*, 1996.

[WEL 97] Wells, David. *Internet Tools Survey – Encryption*. *http://www.objs.com/survey/encrypt.htm.*, 1997.

ANEXOS

Anexo I – Documentação das Classes Desenvolvidas.

Anexo II – Documentação da Aplicação *TN3270*.

Anexo III – Documentação da Aplicação *Telnet*.

Anexo IV – Documentação das Classes *Cryptonite* utilizadas.

ANEXO I – Documentação das Classes Desenvolvidas

Class CsocketS.CSClient

```
java.lang.Object
   |
   +----CsocketS.CSClient
```

public class **CSClient**
extends Object Classe responsável pela transmissão dos dados, no lado do cliente. Contém métodos com a interface semelhante às de uma conexão Socket tradicional.

Author:
Luiz Angelo Barchet Estefanel, lae@inf.ufsm.br

Variable Index

- **conn_factory**
- **CServer**
- **orb**
- **PKG**

Constructor Index

- **CSClient**()

Method Index

- **available**()
  Método que retorna o número de *bytes* que ainda restam para a leitura, no lado do servidor.
- **close**()
  Requisita ao processo servidor para encerrar a conexão com o *host* provedor do serviço.
- **connect**()
  Inicializa o ORB do lado do cliente, e se conecta ao Servidor utilizando o arquivo que contêm a referência IOR (arquivo CsocketS.ref, contém uma referência ao processo do servidor).
- **open**(String, int)
  Inicia uma conexão ao *host* provedor do serviço desejado.
- **read**(byte[])
  Método que recebe os dados encriptados, vindos do servidor.
- **write**(byte[])
  Método para enviar os dados seguros pela rede, como no Socket padrão.

Variables

CServer

```
private CSServer CServer
```

PKG

```
private Packer PKG
```

orb

```
private ORB orb
```

conn_factory

```
private ConnFactory conn_factory
```

Constructors

CSClient

```
public CSClient()
```

Methods

connect

```
public void connect()
```

Inicializa o ORB do lado do cliente, e se conecta ao Servidor utilizando o arquivo que contêm a referência IOR (arquivo CsocketS.ref, contém uma referência ao processo do servidor). Requisita a criação de um processo servidor próprio, para a manutenção da conexão. Também inicia a geração dos *tickets* e das chaves criptográficas

close

```
public void close() throws IOException
```

Requisita ao processo servidor para encerrar a conexão com o *host* provedor do serviço.

open

```
public void open(String host,
                 int port) throws IOException
```

Inicia uma conexão ao *host* provedor do serviço desejado.

Parameters:

host - String que contém o nome do *host* a qual se deseja conectar.
port - Int que indica a porta do serviço desejado (por exemplo, 23 para *telnet*).

read

```
public int read(byte buf[]) throws IOException
```

Método que recebe os dados encriptados, vindos do servidor.

Parameters:

buf - *array* de *bytes* que será preenchido com os dados recebidos.

Returns:

o número de *bytes*armazenados no buffer buf.

write

```
public void write(byte buf[])
```

Método para enviar os dados seguros pela rede, como no Socket padrão.

Parameters:

buf - Buffer preenchido com os dados a serem enviados.

available

```
public int available()
```

Método que retorna o número de *bytes* que ainda restam para a leitura, no lado do servidor. É utilizado por aplicações como o *telnet*, que transmitem e recebem os dados caracter a caracter. Em aplicações como o *TN3270* os dados sempre são transmitidos em grandes blocos (página a página), não havendo a necessidade desse método.

Returns:

o número de *bytes* disponíveis para leitura no buffer do servidor.

Class CsocketS.CSServer_impl

```
java.lang.Object
   |
   +----org.omg.CORBA.portable.ObjectImpl
           |
           +----org.omg.CORBA.DynamicImplementation
                   |
                   +----CsocketS._CSServerImplBase
                           |
                           +----CsocketS.CSServer_impl
```

public class **CSServer_impl**
extends _CSServerImplBase Corresponde ao lado do Servidor da implementação das classes. Recebe as mensagens através das interfaces comuns com o cliente, e coordena as operações no lado do servidor.

Author:
Luiz Angelo Barchet Estefanel, lae@inf.ufsm.br

Variable Index

- **id**
- **inputs**
- **orb_**
- **outs**
- **PKG**
- **socket**

Constructor Index

- **CSServer_impl**(ORB, int)
  Construtor da classe, inicializa o contato com o ORB e dispara a geração das chaves e *tickets*.

Method Index

- **available**()
  Retorna o número de bytes disponíveis para leitura no buffer do serviço.
- **close**()
  Fecha a conexão de rede com o serviço.
- **destroy**()
  Requisita a eliminação desse objeto servidor, quando não é mais útil (a conexão de rede já foi fechada).
- **get_id**()
  Retorna o número identificador desse objeto servidor.
- **getKey**(IarrayHolder)
  Interface para troca de chaves entre o cliente e o servidor.
- **open**(String, int)
  Cria uma conexão *socket* com a máquina provedora do serviço.
- **read**(IarrayHolder)
  Recebe do cliente um *buffer* (IarrayHolder) para ser preenchido com os dados recebidos do serviço.
- **write**(byte[])

Envia, através da conexão *socket* com o serviço os dados recebidos do cliente.

Variables

id

```
private int id
```

socket

```
private Socket socket
```

inputs

```
private BufferedInputStream inputs
```

outs

```
private BufferedOutputStream outs
```

PKG

```
private Packer PKG
```

orb_

```
private ORB orb_
```

Constructors

CSServer_impl

```
public CSServer_impl(ORB orb,
                     int id_)
```

Construtor da classe, inicializa o contato com o ORB e dispara a geração das chaves e *tickets*.

Parameters:

orb - referência ao ORB do Servidor, repassado pelo FactoryServer
id_ - número identificador de ordem do Servidor. Útil para contar quantos servidores já foram ativados.

Methods

get_id

```
public int get_id()
```

Retorna o número identificador desse objeto servidor.

Returns:

inteiro que indica o número de ordem desse objeto.

Overrides:

get_id in class _CSServerImplBase

destroy

```
public void destroy()
```

Requisita a eliminação desse objeto servidor, quando não é mais útil (a conexão de rede já foi fechada).

Overrides:

destroy in class _CSServerImplBase

close

```
public void close()
```

Fecha a conexão de rede com o serviço.

Overrides:

close in class _CSServerImplBase

open

```
public void open(String host,
                 int port)
```

Cria uma conexão *socket* com a máquina provedora do serviço.
Parameters:
host - nome da máquina para qual deve ser aberta a conexão.
port - número indicador da porta de conexão ao serviço.
Overrides:
open in class _CSServerImplBase
read

```
public int read(IarrayHolder rarray)
```

Recebe do cliente um *buffer* (IarrayHolder) para ser preenchido com os dados recebidos do serviço.
Parameters:
rarray - - Objeto do tipo IarrayHolder que contém um *buffer*.
Returns:
rarray - Retorna através de referência
Overrides:
read in class _CSServerImplBase
write

```
public void write(byte warray[])
```

Envia, através da conexão *socket* com o serviço os dados recebidos do cliente.
Parameters:
warray - *array* de *bytes* que contém a mensagem segura.
Overrides:
write in class _CSServerImplBase
getKey

```
public void getKey(IarrayHolder rarray)
```

Interface para troca de chaves entre o cliente e o servidor. A troca dá-se utilizando a estrutura rarray para receber a chave do cliente e enviar a chave do servidor.
Parameters:
rarray - - Objeto IarrayHolder que contém a chave do cliente na forma de *array* de *bytes*. Pela mesma estrutura retorna a chave do servidor.
Overrides:
getKey in class _CSServerImplBase
available

```
public int available()
```

Retorna o número de bytes disponíveis para leitura no buffer do serviço.
Returns:
número de bytes no buffer de envio do serviço.
Overrides:
available in class _CSServerImplBase

Class CsocketS.Packer

```
java.lang.Object
   |
   +----CsocketS.Packer
```

public class **Packer**
extends Object Classe que coordena as operações referentes à adaptação da mensagem para uma estrutura segura para a transmissão. Coordena também a extração da mensagem cifrada recebida. Essas operações são:

- Geração das chaves criptográficas;
- Troca de chaves públicas entre o cliente e o servidor;
- Inicialização e verificação dos *tickets*;
- Serialização e desserialização do pacote de dados (mensagem + *tickets*);
- Assinatura e verificação da origem da mensagem;
- Cifragem e decifragem da mensagem transmitida.

Author:
Luiz Angelo Barchet Estefanel, lae@inf.ufsm.br

Variable Index

- **ECT**
- **myKeys**
- **otherKey**
- **otherSideTicket**
- **SLR**
- **TUTIL**

Constructor Index

- **Packer**()
  Construtor da classe Packer, inicializa alguns parâmetros (*tickets*, chaves criptográficas).

Method Index

- **get_Public**()
  Método utilizado na troca das chaves, entre os processos cliente e servidor.
- **packing**(byte[])
  Monta uma mensagem segura, apta a trafegar na rede.
- **set_Other**(Key)
  Método que armazena a chave pública da outra parte, trocada pela rede.
- **unpacking**(byte[])
  Extrai a mensagem a partir dos dados seguros recebidos via rede.

Variables

myKeys

```
private KeyPair myKeys
```

otherKey

```
private Key otherKey
```

otherSideTicket

`private Vector otherSideTicket`

TUTIL

`private TicketUtil TUTIL`

ECT

`private Encryptool ECT`

SLR

`private Serializador SLR`

Constructors

Packer

`Packer()`

Construtor da classe Packer, inicializa alguns parâmetros (*tickets*, chaves criptográficas).

Methods

unpacking

`public byte[] unpacking(byte message[])`

Extrai a mensagem a partir dos dados seguros recebidos via rede. Executa as seguintes operações, em ordem:

- decifragem dos dados, com a chave privada;
- verificação da assinatura dos dados, com a chave pública do emissor;
- desserialização do pacote de dados;
- verificação dos *tickets*;
- extração da mensagem;

Parameters:

message - ***array* de *bytes*** recebido pela rede, contendo a mensagem cifrada e assinada.

Returns:

array* de *bytes contendo a mensagem original extraída da mensagem segura que veio pela rede.

packing

`public byte[] packing(byte message[])`

Monta uma mensagem segura, apta a trafegar na rede. Executa as seguintes operações, em ordem:

- geração de um novo *ticket* identificador dessa transmissão;
- criação de um pacote de dados contendo a mensagem e os *tickets*;
- serialização do pacote de dados;
- assinatura dos dados, com a chave privada do emissor;
- cifragem dos dados, com a chave pública do receptor;

Parameters:

message - ***array* de *bytes*** contendo a mensagem original.

Returns:

array* de *bytes contendo a mensagem segura, apta ao tráfego na rede.

get_Public

`public Key get_Public()`

Método utilizado na troca das chaves, entre os processos cliente e servidor.

Returns:

Key - chave pública do objeto.

set_Other

```
public void set_Other(Key other)
```

Método que armazena a chave pública da outra parte, trocada pela rede.
Parameters:
other - - Chave pública enviada pelo outro participante da conexão.

All Packages Class Hierarchy This Package Previous Next Index

Class CsocketS.TicketUtil

```
java.lang.Object
   |
   +----CsocketS.TicketUtil
```

public class **TicketUtil**
extends Object Classe responsável pela geração dos *tickets* utilizados para a identificação das conexões e manutenção do *timeout*. Realiza tanto a criação de novos *tickets* quanto a verificação da validade dos *tickets* recebidos.

Author:

Luiz Angelo Barchet Estefanel, lae@inf.ufsm.br

Variable Index

- **myTicket**
- **TicketNumLength**
- **maxTime**

Constructor Index

- **TicketUtil**()
  Construtor do Objeto TicketUtil.

Method Index

- **getTicket**()
  Método para buscar o *ticket* atual.
- **makeTicket**()
  Gera um novo *ticket*, com um identificador randômico e um *timestamp* correspondente ao "tempo" local.
- **testTicket**(Vector)
  Verificação do *ticket* recebido, que corresponde ao ticket gerado por esse objeto, enviado na transmissão anterior.

Variables

myTicket

```
private Vector myTicket
```

TicketNumLength

```
public int TicketNumLength
```

maxTime

```
public int maxTime
```

Constructors

TicketUtil

```
TicketUtil()
```

Construtor do Objeto TicketUtil. Inicializa com a criação de um *ticket* padrão, para fim de

conexão entre as partes.

Methods

makeTicket

```
public void makeTicket()
```

Gera um novo *ticket*, com um identificador randômico e um *timestamp* correspondente ao "tempo" local. Esse *ticket* não é retornado porque deve ser usado tanto na transmissão de um novo *ticket* quanto na verificação do *ticket* recebido.

getTicket

```
public Vector getTicket()
```

Método para buscar o *ticket* atual.

Returns:

ticket - *Ticket* atualmente ativo no sistema.

testTicket

```
public boolean testTicket(Vector ticket)
```

Verificação do *ticket* recebido, que corresponde ao ticket gerado por esse objeto, enviado na transmissão anterior. Compara o número identificador único do *ticket*. Se for semelhante, verifica se o *ticket* não expirou o tempo limite (*timeout*).

Parameters:

ticket - O *ticket* correspondente ao receptor, armazenado junto com a mensagem recebida.

Returns:

true se o *ticket* recebido confere com o *ticket* armazenado, e ainda não expirou o tempo máximo;
false se o *ticket* não confere ou já ultrapassou o prazo limite de tempo.

All Packages Class Hierarchy This Package Previous Next Index

Class CsocketS.Serializador

```
java.lang.Object
   |
   +----CsocketS.Serializador
```

public class **Serializador**
extends Object Classe responsável pela **serializaçãoi** do pacote de dados que contém a mensagem e os *tickets* do cliente e do servidor. Implementado de acordo com a especificação de serialização do JDK 1.1.5.

Author:
Luiz Angelo Barchet Estefanel, lae@inf.ufsm.br

Constructor Index

Serializador()

Method Index

desserializar(byte[])
Realiza a operação de deserialização do *array* de *bytes*, recebido, remontando o *Vector* que contém a mensagem e os *tickets*..

serializar(Vector)
Realiza a operação de serialização do pacote de dados (um *Vector*), transformando-o em um *array* de *bytes* que descreve a estrutura e o conteúdo do pacote.

Constructors

Serializador

```
public Serializador()
```

Methods

serializar

```
byte[] serializar(Vector v)
```

Realiza a operação de serialização do pacote de dados (um *Vector*), transformando-o em um *array* de *bytes* que descreve a estrutura e o conteúdo do pacote.

Parameters:
v - **Vector** - Objeto que contém a mensagem e os *tickets* do cliente e do servidor.

Returns:
array* de *bytes - representa a serialização do *Vector* fornecido na entrada.

desserializar

```
Vector desserializar(byte source[])
```

Realiza a operação de deserialização do *array* de *bytes*, recebido, remontando o *Vector* que contém a mensagem e os *tickets*..

Parameters:
source - ***array* de *bytes*** - descreve o pacote de dados.

Returns:
Vector - contêm a mensagem e os *tickets do cliente e do servidor.*

Class CsocketS.Encryptool

```
java.lang.Object
   |
   +----CsocketS.Encryptool
```

public class **Encryptool**
extends Object Encryptool é a classe que realiza as operações de cifragem e decifragem usadas nos processos de criptografia e assinatura digital.

Author:
Luiz Angelo Barchet Estefanel, lae@inf.ufsm.br

Variable Index

- **keySize**
- **keyType**
- **mailLength**
- **nameLength**

Constructor Index

- **Encryptool**()

Method Index

- **decrypt**(byte[], Key)
  Método de decifragem dos dados.
- **encrypt**(byte[], Key)
  Método de cifragem dos dados.
- **make_key**()
  Gera um par de chaves criptográficas (RSA) com um comprimento de 256 bits.

Variables

keyType

```
String keyType
```

keySize

```
int keySize
```

mailLength

```
int mailLength
```

nameLength

```
int nameLength
```

Constructors

Encryptool

```
public Encryptool()
```

Methods

make_key

```
public KeyPair make_key()
```

Gera um par de chaves criptográficas (RSA) com um comprimento de 256 bits. O nome e o e-mail do proprietário também são gerados aleatoriamente, buscando aumentar segurança das chaves.

Returns:

KeyPair - uma estrutura de dados definida no pacote criptográfico *Cryptonite*.

encrypt

```
public byte[] encrypt(byte message[],
                      Key chave)
```

Método de cifragem dos dados. Utiliza os métodos de cifragem estabelecidos no pacote criptográfico *Cryptonite*. Pode ser utilizada para criptografia, ao fornecer uma chave pública, ou para a assinatura digital, fornecendo uma chave privada.

Parameters:

message - - Mensagem a ser cifrada.

chave - - Chave privada ou pública para efetuar uma assinatura ou cifragem, respectivamente.

Returns:

***array* de *bytes* - A mensagem cifrada, após o processo.**

decrypt

```
public byte[] decrypt(byte message[],
                      Key chave)
```

Método de decifragem dos dados. Utiliza os métodos de cifragem estabelecidos no pacote criptográfico *Cryptonite*. Pode ser utilizada para decriptografia, ao fornecer uma chave privada, ou para a verificação de uma assinatura digital, fornecendo uma chave pública.

Parameters:

message - - Mensagem a ser decifrada.

chave - - Chave pública ou privada para verificar uma assinatura ou decifrar uma mensagem, respectivamente.

Returns:

***array* de *bytes* - A mensagem decifrada, após o processo.**

All Packages Class Hierarchy This Package Previous Next Index

ANEXO II – Documentação da Aplicação TN3270

Homepage: http://www.unige.ch/hotjava/

TN3270 JAVA EMULATOR AND LIBRARIES

(Software awarded during the SUN Microsystems 1995 JAVA Contest)

- ## WHAT IS IT?

The TN3270 JAVA EMULATOR AND LIBRARIES is a software package intended to :

1. create a universal IBM-3270 emulator;
2. create a user-friendly programming package (Libraries) to handle 3270 sessions;
3. create a graphical interface to query the SIBIL catalog.

The TN3270 JAVA Emulator, applications running on IBM mainframes may now be accessed from any Internet-Intranet JAVA browser.

Thanks to the TN3270 JAVA Libraries, extensive mouse navigation in an intuitive graphical interface including useful features such as memorization of previous searches and explored paths is now available.

- ## HOW IT WORKS?

HotSIBIL is a lively example of what the Java TN3270 libraries and emulator can do for Intranet-Internet TN3270 sessions.
HotSIBIL provides access to SIBIL catalog, one of the largest bibliographic databases in Switzerland.
SIBIL is TN3270 application available in a traditional text based environement which was created some twenty years ago, and is in wide use today as it references over 2 million titles.

How to use HotSIBIL, or more blah-blah about it.

DOWNLOAD IT NOW!

You may see and try it by yourself. Please read and accept the terms applicable for the use or modification of the software.

- A simple example using the Tn3270 Java libraries.

Class tn3270.Tn3270

```
java.lang.Object
   |
   +----tn3270.Tn3270
```

public class **Tn3270**
extends Object Defining an applet that uses IBM-3270 sessions:

```
import  tn3270.*;
public class Java3270 extends Applet implements Tn3270paint {
}
```

Constructor Index

- **Tn3270**(int, Tn3270paint)

Method Index

- **closeConnection**()
- **eraseCursor**()
- **insertString**(String)
- **keyBackwardTab**()
- **keyChar**(int)
- **keyClear**()
- **keyDelete**()
- **keyDeleteAllInput**()
- **keyDeleteEndOfField**()
- **keyDown**()
- **keyEnter**()
- **keyFieldMark**()
- **keyForwardTab**()
- **keyHome**()
- **keyInsertMode**(boolean)
- **keyLeft**()
- **keyNewLine**()
- **keyPA**(int)
- **keyPF**(int)
- **keyReset**()
- **keyRight**()
- **keyRubout**()
- **keyToggleMode**()
- **keyUp**()
- **openConnection**(String, int)
- **processInputStream**(short[], int)
  process incoming data
- **readInputStream**(short[])
  read incoming data from IBM-3270 sessions
- **repaintCursor**()
- **repaintScreen**()
- **setCursorPosition**(int)

Constructors

Tn3270

```
public Tn3270(int modelNumber,
              Tn3270paint paint)
```

Parameters:

modelNumber - defines the 3270 screensize: (2: 24x80), (3: 32x80), (4: 43x80), (5: 27x132)

paint - represents the current Applet. The Tn3270paint class defines an interface for the Tn3270 in order to call back the necessary painting fonctions defined in the applet.

Methods

closeConnection

```
public void closeConnection() throws IOException
```

openConnection

```
public void openConnection(String host,
                           int port) throws IOException
```

Parameters:

host - TCP-IP IBM-3270 address

port - TCP-IP port

processInputStream

```
public void processInputStream(short netBuf[],
                               int netBufLen) throws IOException
```

process incoming data

readInputStream

```
public int readInputStream(short buf[]) throws IOException
```

read incoming data from IBM-3270 sessions

keyBackwardTab

```
public void keyBackwardTab()
```

keyDown

```
public void keyDown()
```

keyForwardTab

```
public void keyForwardTab()
```

keyHome

```
public void keyHome()
```

keyLeft

```
public void keyLeft()
```

keyNewLine

```
public void keyNewLine()
```

keyRight

```
public void keyRight()
```

keyUp

```
public void keyUp()
```

setCursorPosition

```
public void setCursorPosition(int addr)
```

Parameters:

addr - set cursor position in respect of top-left corner

keyClear

```
public void keyClear() throws IOException
```

keyEnter

```
public void keyEnter() throws IOException
```

keyPA

```
public void keyPA(int no) throws IOException
```

keyPF

```
public void keyPF(int no) throws IOException
```

keyDelete

```
public void keyDelete()
```

keyDeleteAllInput

```
public void keyDeleteAllInput()
```

keyDeleteEndOfField

```
public void keyDeleteEndOfField()
```

keyRubout

```
public void keyRubout()
```

insertString

```
public void insertString(String str)
```

keyChar

```
public void keyChar(int c)
```

keyFieldMark

```
public void keyFieldMark()
```

keyInsertMode

```
public void keyInsertMode(boolean on)
```

keyToggleMode

```
public void keyToggleMode()
```

keyReset

```
public void keyReset()
```

eraseCursor

```
public void eraseCursor()
```

repaintCursor

```
public void repaintCursor()
```

repaintScreen

```
public void repaintScreen()
```

ANEXO III – Documentação da Aplicação Telnet

Homepage: http://www.first.gmd.de/persons/leo/java/Telnet/

The Java(tm) Telnet Applet: Documentation

© 1996-98 Matthias L. Jugel, Marcus Meißner

The package contains several parts of which the most important one is the **Telnet Applet/Application**. Select from the list below what you are interested in. If you only want to use the applet choose Telnet from **Setup** and if you want to use the packages in your own programming, select the appropriate from **Source Code**.

READ THIS FIRST

Setup

[Telnet | Terminal Emulation | Modules]
[Applet Wrapper | Proxy Server]

Source Code

[Telnet | Terminal Emulation | Modules]
[Applet Wrapper | Proxy Server]
[Packages | Field and Method Index | Class Tree]

Get the latest version here!

READ THIS FIRST

We found that some people have no knowlegde whatsoever of java and its restrictions. We have compiled a few questions and answers here as well as some reasons why you should or should not use **The Java(tm) Telnet Applet**:

Some web page told me I need telnet, is this it?

Yes and No! The Applet is a fully featured Telnet and Terminal emulator, but usually you're better off using the program that comes with your system. Most of the UNIX based systems have very good terminal emulators (xterm) and alway have a telnet application. Windows 95 comes with a telnet if you have the network stuff installed it's there: c:\windows\telnet.exe. It should be sufficient. If you want better terminal emulation and *colours* the better choice is **The Java(tm) Telnet Applet**!

I cannot connect to some.where.com? It only says "Trying some.where.com ..."

A Java applet is restricted in several ways. One of the restrictions is that it may *only* connect to the web server where it was downloaded from! So if you put the applet on *www.where.com* but set the "address" field to *some.where.com* you won't get a connection. Read about our relayd daemons!

But I loaded the HTML file from my harddisk and it still does not work!

Netscape and Internet Explorer do not accept your hard disk as secure space. So they will prevent the applet from accessing any resource, such as the network. You may overcome that by adding the directory where the applet is stored to your "CLASSPATH" environment variable *before* running the browser.

Setup Documentation

How to setup the Telnet Applet

Make sure, you got the latest version of the Java(tm) Telnet Applet. Refer to the download page on how to get it and how to extract the files from the archive. After successfully extracting the complete package you should have a directory **Telnet/** containing ***.html**, ***.java** and ***.class** files as well as the directores **Documentation/**, **display/**, **modules/**, **socket/** and **tools**.

To install the applet on your web page you need as least the following files and directories. Make sure that all files and directories are **readable by other users**!

```
index.test.html
telnet.class
appWrapper.class
display/
display/SoftFont.class
display/CharDisplay.class
display/Terminal.class
display/TerminalHost.class
display/vt320.class
socket/
socket/TelnetIO.class
socket/StatusPeer.class
modules/
modules/Module.class
modules/Script.class
modules/ButtonBar.class
modules/MudConnector.class
```

Now edit **index.test.html** to adapt it to your needs or look at the example below! The file is documented and if you have questions about the *Modules* refer to the Module Documentation or look at the Source Code. You will find, that not **telnet.class** is loaded as applet, but **appWrapper.class** instead. This is necessary to enable the **detach** feature!

Important Note:

> We would appreciate to see **credits** on a page using the applet which includes a link to the *applets home page* and names of the *authors* as mentioned on top of this page. You may simply use our test page and edit it to your needs.
> In response we will include a link to your page on our *user page*, if you like.

The telnet applet can be customized using the following parameters:

```
<PARAM NAME=address    VALUE="tanis.first.gmd.de">
<PARAM NAME=port       VALUE="23">
<PARAM NAME=emulation  VALUE="vt320">

<PARAM NAME=proxy      VALUE="www.first.gmd.de">
<PARAM NAME=proxyport  VALUE="31415">
```

The ***proxy*** and ***proxyport*** parameters may be left out. They are needed if your target host is NOT the same as your web server and you are using the relay daemon.

Example:

(all possible parameters)

```
<APPLET CODE="appWrapper.class" WIDTH=600 HEIGHT=480>

<!-- appWrapper parameters -->
<PARAM NAME="applet"      VALUE="telnet">

<!-- optional (WIDTH and HEIGHT should be changed!) -->
<PARAM NAME="startButton" VALUE="Connect to www.first.gmd.de!">
<PARAM NAME="stopButton" VALUE="Shutdown telnet!">
<PARAM NAME="frameTitle" VALUE="The Java Telnet Applet: WWW">

<!-- applet parameters: address and port and emulation -->
<PARAM NAME="address"     VALUE="www.first.gmd.de">
<PARAM NAME="port"        VALUE="23">
<PARAM NAME="emulation"   VALUE="vt320">

<!-- terminal emulation parameters (optional)-->
<PARAM NAME="VTscrollbar" VALUE="true">
<PARAM NAME="VTresize"    VALUE="font">
<PARAM NAME="VTfont"      VALUE="Courier">
<PARAM NAME="VTfontsize"  VALUE="13">
<PARAM NAME="VTid"        VALUE="vt220">
<PARAM NAME="VTcharset"   VALUE="ibm">
```

```
<!-- module parameters: #1 is a buttonbar (optional) -->
<PARAM NAME="module#1"    VALUE="ButtonBar@North">
<PARAM NAME="1#Button"    VALUE="connect|\$connect()">
<PARAM NAME="2#Button"    VALUE="disconnect|\$disconnect()">
<PARAM NAME="3#Button"    VALUE="Detach/Delete Window|\$detach()">
<PARAM NAME="4#Button"    VALUE="Send:|\@send@\r\n">
<PARAM NAME="5#Input"     VALUE="send#10|who">

<!-- module parameter: #2 is a scripting module (optional) -->
<PARAM NAME="module#2"    VALUE="Script">
<PARAM NAME="script"      VALUE="login:|leo">

<!-- make sure, non-java-capable browser get a message: -->
<B>
Your Browser seems to have no Java
support. Please get a new browser or enable Java to see this applet!
</B>
</APPLET>
```

Setting up the Terminal Emulation

The Terminal Emulation is a very important part of the Telnet Applet, because it enables you to use programs that make use of certain features of hardware terminals like **VT100** or **ANSI**. Supplied with the package is an almost **VT320 compliant** terminal emulation, that should include the two mentioned earlier. This means that the applet can do colors, even if the original **VT320** terminal cannot!
The applet supports the ***special graphical character set*** of VT terminals. The new implementation supports all graphical characters with a small drawback. The more graphical characters on the screen the slower is the display. We will remove the current implementation when full UNICODE support is available from all browsers (full JDK 1.2 compatibility).
To configure the terminal emulation look at the list of parameters below:
Note: Default values are typeset in *italics* and other possible values in **bold**.

<PARAM NAME="localecho" VALUE="*auto*">
: Sets the mode the local echo should be handled. If using auto, or if this parameter is not present, the applet autodetects localecho mode using telnet option negotiation. If set to *no*, nothing will be echoed, ever. Any other value enables every character to be echoed.

<PARAM NAME="VTcolumns" VALUE="*80*">
: Sets the columns of the terminal initially. If the parameter VTresize is set to **screen** this may change, else it is fixed.

<PARAM NAME="VTrows" VALUE="*24*">
: Sets the rows of the terminal initially. If the parameter value of VTresize **screen** this may change!

<PARAM NAME="VTfont" VALUE="*Courier*">
: Sets the font to be used for the terminal. It is recommended to use *Courier* or at least a fixed width font.

<PARAM NAME="VTfontsize" VALUE="*14*">
: Sets the font size for the terminal. If the parameter value of VTresize is set to **font** this may change!

<PARAM NAME="VTresize" VALUE="*font*">
: This parameter determines what the terminal should do if the window is resized. The default setting ***font*** will result in resizing the font until is matches the window best. Other possible values are **none** or **screen**. **none** will let nothing happen and **screen** will let the display try to change the amount of rows and columns to match the window best.

<PARAM NAME="VTscrollbar" VALUE="*false*">
: Setting this parameter to **true** will add a scrollbar west to the terminal. Other possible values include **left** to put the scrollbar on the left side of the terminal and **right** to put it explicitely to the right side.

<PARAM NAME="VTid" VALUE="*vt320*">
: This parameter will override the terminal id *vt320*. It may be used to determine special terminal abilities of VT Terminals.

<PARAM NAME="VTbuffer" VALUE="*xx*">
: Initially this parameter is the same as the VTrows parameter. It cannot be less than the

amount of rows on the display. It determines the available scrollback buffer.

<PARAM NAME="VTcharset" VALUE="*none*">

Setting this parameter to **ibm** will enable mapping of ibm characters (as used in PC BBS systems) to UNICODE characters. Note that those special characters probably won't show on UNIX systems due to lack in X11 UNICODE support.

<PARAM NAME="VTvms" VALUE="*false*">

Setting this parameter to **true** will change the Backspace key into a delete key, cause the numeric keypad keys to emit VT100 codes when Ctrl is pressed, and make other VMS-important keyboard definitions.

<PARAM NAME="F*nr*" VALUE="*string*">

Function keys from *F1* to *F20* are programmable. You can install any possible string including special characters, such as

\e	Escape	\b	Backspace	\n	Newline	\r	Return	\xxxx	Character xxxx (decimal)

Please look at the example above.

Setting up Modules

Another feature of the Java(tm) Telnet Applet is the ability to dynamically load **modules**. A module is a java class that is loaded after the applet has been initialized and may be used to *enhance* the user interface or to background work in some way.

To load a module a special parameter has to be added to the applet PARAM tags:

<PARAM NAME=module#*number* VALUE="*modulename*@*direction*">

- ***number*** is a sequence number, used by the applet to determine the modules. Numbers must be adjacent or modules may not be loaded.
- ***modulename*** is the name of the modules to be loaded. Modules already in the package are described below.
- ***@direction*** is the position of the applet in relation to the window. Possible values are: **North, South, East, West**. The module will then be placed accordingly. It is *not* possible to place two modules at the same position! *The positional parameter may be left out and the module will then be placed **North**.*

At the moment the package features three modules:

ButtonBar

The ButtonBar is a modules to enhance the user interface. Using PARAM tags **buttons** and **input fields** can be added to send text to the remote host or to **detach** the applet.

Script

Sometimes it is useful to have simple script abilities. This module **executes a script** based on text received from the remote host.

MudConnector

This module is a special program for the Mud Connector by Andrew Cowan. It loads a list of muds and displays information like host and port.

The ButtonBar

The ButtonBar may be used to add buttons to the applet that execute functions or simply send a specified text to the remote host. In addition it is possible to specify input fields as external input means.

To load the module include the following tag into the **.html** file (example):

<PARAM NAME=module#*1* VALUE="*ButtonBar*">

Below is a description of possible PARAM tags and a description of supported functions:

Buttons:

<PARAM NAME=***number***#Button VALUE="***buttontext***|***buttonaction***">

number is the sequence number and determines the place of the button on the row.
buttontext is a string displayed on the button.
buttonaction may be one of the following functions or strings
(*Note:* the backslash character in front of the dollar sign is mandatory!)

- *simple text* to be sent to the remote host. Newline and/or carriage return characters may be added in C syntax **\n** and **\r**. To support unimplemented function keys the **\e** escape character may be useful. The **\b** backspace character is also supported. The text may contain ***field reference(s)***.
- \$connect(***host***[,***port***]) tries to initiate a connection to the ***host*** at the ***port***, if given. The standard port is 23. ***host*** and ***port*** may be hostname and number or

field reference(s). If a connection already exists nothing will happen.
(*Note:* It is not allowed to have spaces anywhere inside the parenthesis!)

- `\$disconnect()` terminates the current connection, but if there was no connection nothing will happen.
- `\$detach()` detaches the applet from the web browser window and creates a new frame externally. This may be used to allow users to use the applet while browsing the web with the same browser window.
(*Note:* You need to load the applet via the Applet Wrapper or it will not work properly!)

Examples:
(*Note:* It makes sense if you look at the examples for input fields below.)

```
        <PARAM NAME=1#Button VALUE="HELP!|help\r\n">
        <PARAM NAME=2#Button VALUE="HELP:|help \@help@\r\n">
        <PARAM NAME=4#Button VALUE="simple|\$connect(localhost)">
        <PARAM NAME=5#Button VALUE="complete|\$connect(www,4711)">
        <PARAM NAME=6#Button VALUE="connect|\$connect(\@address@)">
        <PARAM NAME=8#Button VALUE="connect to
port|\$connect(\@address@,\@port@)">
        <PARAM NAME=10#Button VALUE="window|\$detach()">
```

Input fields

`<PARAM NAME=`***`number`***`#Input VALUE="`***`fieldname`***`[#`***`length`***`]|`***`initial text`***`[|`***`action`***`]">`

number is the sequence number and determines the place of the field on the row.
fieldname is a symbolic name to reference the input field. A reference may be used in ***button actions*** and is constructed as follows: `\@`***`fieldname`***`@` The **\@fieldname@** macro will be replaced by the string entered in the text field.
length is the length of the input field in numbers of characters.
initial text is the text to be placed into the input field on startup
action may be used similar to a ***button action***. This action will be used if the users presses Return in the inputfield. Leave empty if you only want to use a button to send the text!

Examples:
(*Note:* It makes sense if you look at the examples for buttons before.)

```
<PARAM NAME=3#Input VALUE="help#10|">
<PARAM NAME=7#Input VALUE="address|www.first.gmd.de">
<PARAM NAME=9#Input VALUE="port#5|4711">
```

The Script Module

The script module gives a very simple implementation of an *input triggered* script executor. This means it sends text to the remote host when the received text matches a *pattern* that can be programmed. It executes each *pair of pattern and text* only once and stops working after all patterns have been matched. It will start working again upon a new connection.
To load the module include the following tag into the **.html** file (example):
`<PARAM NAME=module#`*`1`*` VALUE="`*`Script`*`">`
Below is a description of possible PARAM tags and a description of script:

Scripts:

`<PARAM NAME=script VALUE="`***`pattern`***`|`***`text`***`|`***`...`***`">`

A script contains of pairs of *pattern* and *text* strings. If the pattern is matched against the output from the remote host, the corresponding text will be sent. Each pattern will match only **once** per connected session. Thus it is possible to program an autologin as follows: **`"login:|leo|Password:|mypassword|leo@www|ls"`** Newlines will be added automatically to the string sent! At the moment the order of the pattern and text pairs is *not* relevant.
It is possible to prompt the user for input if a match occurs. If the corresponding *text* is a string enclosed in braces ([] or {}) a dialog window is opened with *text* as prompt. A special case is an empty prompt in which case the *pattern* will be shown as prompt. "[Your name:]" would open a dialog window with the text "Your name" as prompt. Curly

braces have a special meaning; any user input will be shown as "*" which makes it possible to program password prompts. Example: "{Your password:}".
A special match like: "login:|[]" can be used to open a dialog and display "login:" as prompt. This works for "{}" as well.

MudConnector

This module is a special edition for the Mud Connector. It features a list of MUDs and a few buttons to connect, disconnect and get infos about the MUDs. A very nice example for a specialized module. To load the module include the following tag into the **.html** file (example):
`<PARAM NAME=module#1 VALUE="MudConnector">` The MudConnector expects the following PARAM tags:

Mudlist URL:

`<PARAM NAME=mudlist VALUE="URL">`

The URL should be a file containing line by line the *MUD name*, the *Mud address* and the *MUD port*, separated by tabulators. The first line in the file should be the number of MUDs in the file.

Example:

`<PARAM NAME=mudlist VALUE="http://www.mud.de/~leo/mudlist.data">`

The Applet Wrapper Setup

The applet wrapper is an applet that does nothing else than loading the, for example, telnet applet. To understand why this is necessary you have to look at the experiences we have made.
Simply using the telnet applet in the following manner:
`<APPLET CODE="telnet.class" WIDTH=600 HEIGHT=480>` and using the detach function stops the applet after you have detached the applet and want to browse the web again. It seems that the Web browser stops all threads connected to the applet if you leave the page where the applet is located and thus it doesn't even update its display anymore.
We have found out that this is not true for applets loaded within the applet on the page (e.g. the appWrapper). Look at the following description on how to setup the appWrapper. It will probably work with any given applet out on the web!

```
<APPLET CODE="appWrapper.class" WIDTH=600 HEIGHT=480>
<PARAM NAME=applet VALUE="telnet>

<!-- optional (WIDTH and HEIGHT should be changed!) -->
<PARAM NAME=startButton VALUE="text">
<PARAM NAME=stopButton VALUE="text">
<PARAM NAME=frameTitle VALUE="text">
<!-- all other telnet applet parameters go here -->
</APPLET>
```

The **appWrapper** knows only about the PARAM tag **applet**, which is the applet to be loaded. In this case it must be in the same directory as the **appWrapper.class**. Refer to the telnet example above for the telnet parameters.
If a **startButton** is specified the applet won't start automatically, but instead the appWrapper will display a button with the *text* on it. The **stopButton** defines the text that appears on the button when the applet is loaded and running and **frameTitle** specifies the frame title text of the window the applet runs in.

Setting up the proxy server

There are two proxy servers provided with the telnet applet. The first one is written in java and does support connections to **one** target host only and the second one is written in C and supports different targets (called relay daemon).

[Java Proxy | Simple Relay Daemon | Relay Daemon]

The Java Proxy Server

The proxy server is a small java program to overcome the security restrictions of java capable web browsers.
The proxy is used to redirect a connection to a given host. Usually an applet can only connect to the web server it has been loaded from. Installing the proxy on your web server allows the applet to connect to a host you would like to connect to.

How to run the proxy application?

To run the proxy you require the following:

1. A java interpreter (usually included in the JDK)
2. A compiled version of the proxy (proxy.class)

On the *WWW-Server command line* run the proxy server as follows: `java proxy 9999 remotehostname 23` This lets the proxy listen on port 9999 and it redirects all connections to the host "remotehostname" at port 23. You can leave the port parameter out if it is 23 (telnet port).
The proxy should start with something like the output below: `proxy: destination host is remotehostname at port 23 proxy: listening on port 9999` Upon successful connection the output should produce something like this: `proxy: accepted connection from augra.first.gmd.de proxy: connecting www.first.gmd.de <-> remotehostname`

How to shut down the proxy?

To shut down the proxy press ^C (Ctrl+C or Strg+C on a german keyboard) if you have startet it normally. More advanced users will run the proxy like `java proxy 9999 remotehost 23 & errorlog &` to put it into the background. The "errorlog" file should then contain any messages. You can kill that process by looking for the process id (ps | grep proxy) and issuing the kill <processid> command (this applies to UNIX only).

I get an error message like "class proxy not found"!

You may have to set the CLASSPATH environment variable to point to the current directory or to the directory where proxy.class is located.

The Simple Relay Daemon

The *simple* relay daemon works just like the proxy above, but is a C version. It allows connections only to a specified host and port which is preferrable for security reasons. You can run the program (after compiling it) with the following command line: `relayd serverport targethost targetport or just relayd serverport targethost` It will then listen on the *serverport* of the machine you started it and connect to the *targethost*. The standard *targetport* is 23.

The Relay Daemon

The relay daemon is a program written by Marcus Meißner to support different target hosts. It relays the connections from the applet to a host that must be given to the relay daemon after connecting.
The daemon expects a string `relay` *`address port`* It must be run on the web server of the applet. The relay daemon is not included in compiled form, because we would have to support a number of platforms. However, you can write to us if you need a special compiled version for your hardware platform.
You should include the following tags to tell telnet that it is supposed to use the prox server

```
<PARAM NAME=proxy      VALUE="www.first.gmd.de">
<PARAM NAME=proxyport  VALUE="31415">
```

Source Code Documentation

The Source Code of **The Java(tm) Telnet Applet** is available under the terms of the **GNU General Public License** as documented in the file COPYING. In case you would like to use the packages as libraries please apply the **GNU Library General Public License** as documented in the file COPYING.LIB.
Select from the structure below, what you would like to see. Each file contains a **Version:** field determining its current status and version. If you are not sure to have the most current version, please look here.
If you are unsure, whether you've got the newest version, compare your copy of the file **REVISION** and this **REVISION**, which is a direct link to the home page. The latest changes are documented in the file **CHANGES**.

- appWrapper.java
- telnet.java
    - Socket Package
        - socket/TelnetIO.java
        - socket/StatusPeer.java
    - Display Package
        - display/CharDisplay.java
        - display/SoftFont.java
        - display/Terminal.java
        - display/TerminalHost.java

        - display/vt320.java
    - Module Package
        - modules/Module.java
        - modules/ButtonBar.java
        - modules/Script.java
        - modules/MudConnector.java
- IOtest.java
- CharDisplayTest.html
- Tools
    - tools/proxy
    - tools/redirector (proxy.java)
    - tools/relayd.c (simple proxy)
    - tools/mrelayd.c (enhanced proxy)

Get the latest version here!
Last modified: Wed Jul 23 14:55:15 1997 by Matthias L. Jugel

ANEXO IV – Documentação do Pacote Cryptonite

Homepage: http://www.hi.is/~logir/

Cryptonite Java Package

What is it?

Cryptonite is a non-certified 100% pure java library for using strong encryption in your java 1.1 programs. It will eventually include all the various key management tools including key certificates, and even a key-server for automatic exchange of keys. It is distributed with full source-code, since noone can be expected to trust an encryption package without seeing the source.
However, Cryptonite is still in a pre-beta form with most of the key-management features untested, although key generation and encryption/decryption is supported.

What does it cost?

Well, that depends on what you wish to do with it. If you write software for your own private use, in a non-commercial environment, it is free.
If you write software and distribute it freely and are not compensated for your work, it is free.
If you release a program which uses Cryptonite and which is in some way sold, either as shareware, shrinkwrap, or otherwise, you must pay the copyright holder (that's me) the price of one copy of the program or USD100, whichever is less.
If you write software, which is not sold, but are compensated for this work, for example for writing in-house software, you must buy a license for USD50.
However, it will be free as long as I still consider it to be a pre-release.

Is it legal?

That's a difficult question. Cryptonite is developed in Iceland, which has no laws about encryption whatsoever. This means that the use in and export from Iceland is clearly legal.
Most countries allow the use of encryption software within their borders, but do not allow export. This includes the US and most of Europe. Some countries do not allow the use of encryption software at all. These tend to be run by governments which are afraid of their own people.
I'm afraid you will have to check the legal status of encryption software in your region if you are unsure.

How does it work?

Cryptonite is a framework for both private and public key encryption and stream classes to easily encrypt and decrypt all data flowing through them. This means that using strong encryption becomes about as easy as using the compression streams included in JDK 1.1. For information about particular encryption algorithms you should look at the RsaKey and CaesarKey classes in the javadoc documentation.
Cryptonite also includes a lot of helper classes, including fingerprinting and digital signatures.

How secure is it?

As far as I know, the RSA algorithm should be next to unbreakable, or about as difficult to break as any modern encryption. It all depends on the size of the key you use and the quality of the random number generator used when creating it.
The random number generator included in Cryptonite is fair, but not perfect. It passes the Chi-square test and the gap test as often as you would expect a truly random stream of bytes to do when running in Linux, but slightly less often when running on OS/2. The implementation of the random number generator is such that it may behave differently on different systems and concievably could behave quite predictably on some.
You should run the TestRandom program on your system to see if it behaves well on your system.

How do I use it?

There are two ways to encrypt data with Cryptonite. One is to encrypt an array of bytes using the `byte[] Key.encrypt(byte[])` method and decrypt it with the `byte[] Key.decrypt(byte[])` method.
A more useful method is to use the `EncryptStream` and `DecryptStream` classes to filter you i/o operations. They will encrypt or decrypt all data that passes through them.
You can also have a look at the javadoc documentation.

How do I get it?

Here you may view the license agreement and possibly download version 0.08 of Cryptonite. If you use Cryptonite in one of your projects, I'd like to hear about it. If you decide not to use Cryptonite I'd also like to hear why, so that I may improve it.

Class IS.hi.logir.cryptonite.Key

```
java.lang.Object
   |
   +----IS.hi.logir.cryptonite.Key
```

public abstract class **Key**
extends Object
implements Serializable, Fingerprintable This class is the ancestor of classes to hold one public or private key. It's descendants also handle the actual encryption and decryption of data.
Subclasses of Key should be "read-only" in the same sense as the String class and not implement any methods to change the state of the object.
The format of the ciphertext is as follows:

> **cipher block**=<decrypted size><cipher chunk>[cipher chunk[... [cipher chunk]...]]
> Each cipher chunk is a whole number of bytes. The excact number of bytes depends on the key used to encrypt the data.
> There may not be enough plain data to fill the last chunk, in which case the last chunk is calculated as if 0's were added to the end of the plaintext.

Author:
> Logi Ragnarsson (logir@hi.is)

See Also:
> KeyPair, KeyRing

Variable Index

- **fingerprint**
  The key's MD5 fingerprint.
- **isRegistered**
- **keyClasses**
- **ONE**
  The constant one.
- **ownerMail**
  The key-owner's e-mail address.
- **ownerName**
  The key-owner's name.
- **primeCertainty**
  We allow a chance of 0.5**primeCertainty chance that the primes we generate are in fact composite numbers.
- **random**
  Random is used by descendants of this class to generate random numbers.
- **signatures**
  The signatures vector may contain certificates for this key.
- **ZERO**
  The constant zero.

Constructor Index

- **Key**(String, String)
  Create a new key.

Method Index

addKeyClass(String, Class)
Add the class c to the list of available key classes.

algorithmName()
Return the name of the algorithm used by this key.

blockSize()
Returns a number n, such that least space will be wasted if a mutiple of n bytes is encrypted at once.

calcFingerprint()
Return the key's MD5 fingerprint.

create(String, int, String, String)
Create a new KeyPair for the named key class with the specified size.

decrypt(byte[])
Decrypt the data from source with this key and return an array with the plain bytes.

decrypt(Data, Data)
Decrypt the data from source with this key and put the plain data in dest.

encrypt(byte[])
Encrypt source with this key and return an array with the encrypted data.

encrypt(byte[], int)
Encrypt the first sourceLength bytes of source with this key and return an array with the encrypted data.

encrypt(Data, Data)
Encrypt source with this key and put the result in dest.

getFingerprint()
Return the key's MD5 fingerprint.

getOwnerMail()
Return the e-mail address of the owner.

getOwnerName()
Return the name of the owner.

getSignatures()
Return an enumeration of this keys certificates.

getSize()
Return the "size" of the key.

hashCode()
Return a hash-code based on the MD5 fingerprint.

matches(Key)
Returns true if this and key are a matched pair of public/private keys.

registerIncludedClasses()
Register all the included classes so they can be created by name.

toString()
Return a string representation of this key.

Variables

keyClasses

```
private static Hashtable keyClasses
```

isRegistered

```
private static boolean isRegistered
```

random

```
public static Random random
```

Random is used by descendants of this class to generate random numbers. It should be a random number without a period, which rules out all generators based on iterated

functions, such as java.util.Random.
If this field is null, an instance of RandomSpinner will be created and used.

See Also:
RandomSpinner

primeCertainty

```
public static int primeCertainty
```

We allow a chance of 0.5**primeCertainty chance that the primes we generate are in fact composite numbers.
The default value of 32 will give a false prime less than once in every 4,000,000,000 tries.
However, finding one prime often takes several tries, but the likelyhood of a false prime being returned is still negligeble.

ONE

```
protected static final BigInteger ONE
```

The constant one.

ZERO

```
protected static final BigInteger ZERO
```

The constant zero.

ownerName

```
private String ownerName
```

The key-owner's name.

ownerMail

```
private String ownerMail
```

The key-owner's e-mail address.

fingerprint

```
private transient Fingerprint fingerprint
```

The key's MD5 fingerprint. It is never stored with the key, but re-calculated at run-time.
fingerprint may be null in which case it needs to be re-calcualted.

signatures

```
private Vector signatures
```

The signatures vector may contain certificates for this key.
Never sign a key unless you have first-hand knowledge that it is valid!
A key which is not certified by a trusted party should not be used.

See Also:
Signature

Constructors

Key

```
public Key(String ownerName,
           String ownerMail)
```

Create a new key. It will be marked as belonging to ownerName with e-mail address ownerMail.

Methods

addKeyClass

```
public static void addKeyClass(String name,
                               Class c)
```

Add the class c to the list of available key classes. This allows the creation of keys of this class by name.
All regstered classes must be sub-classes of the Key class and implement a `static KeyPair`

`createKeys(int size, String ownerName, String ownerMail)` method similar to that in the RsaKey class.

See Also:
create, createKeys

registerIncludedClasses

```
private static void registerIncludedClasses() throws BadKeyClassException
```

Register all the included classes so they can be created by name.

create

```
public static KeyPair create(String name,
                             int size,
                             String ownerName,
                             String ownerMail) throws BadKeyClassException
```

Create a new KeyPair for the named key class with the specified size.
A key class with the specified name must previously have been registered by calling addKeyClass. This is automatically done for those classes included with cryptonite but any other classes must be registered manually.
The meaning of the size parameter depends on the key class. If more than one parameter is needed to fully specify the kind of key to create, a default will be used. If this is not acceptable, use the constructor for the particular key class.
The keys will be marked as belonging to ownerName with e-mail address ownerMail.

See Also:
addKeyClass

getSize

```
public abstract int getSize()
```

Return the "size" of the key. This is a measure of how difficult it is to break and is heavily dependant on the algorithm used. In many cases it will be the actual number of bits needed to store the key.

getOwnerName

```
public final String getOwnerName()
```

Return the name of the owner.

getOwnerMail

```
public final String getOwnerMail()
```

Return the e-mail address of the owner.

calcFingerprint

```
protected abstract Fingerprint calcFingerprint()
```

Return the key's MD5 fingerprint.

See Also:
MD5State, Fingerprintable

getFingerprint

```
public final Fingerprint getFingerprint()
```

Return the key's MD5 fingerprint.

getSignatures

```
public Enumeration getSignatures()
```

Return an enumeration of this keys certificates. These are of the Signature class.

See Also:
Signature

algorithmName

```
public abstract String algorithmName()
```

Return the name of the algorithm used by this key.

toString

```
public String toString()
```

Return a string representation of this key.

Overrides:

toString in class Object

hashCode

```
public int hashCode()
```

Return a hash-code based on the MD5 fingerprint.

Overrides:

hashCode in class Object

matches

```
public abstract boolean matches(Key key)
```

Returns true if this and key are a matched pair of public/private keys.

blockSize

```
public int blockSize()
```

Returns a number n, such that least space will be wasted if a mutiple of n bytes is encrypted at once.

encrypt

```
public abstract void encrypt(Data source,
                             Data dest)
```

Encrypt source with this key and put the result in dest. The source.buf and dest.buf arrays must be different.

encrypt

```
public final byte[] encrypt(byte source[],
                            int sourceLength)
```

Encrypt the first sourceLength bytes of source with this key and return an array with the encrypted data. This method ultmately calls encrypt(Data,Data), but causes more memory allocation and garbage collection than using that method directly.

encrypt

```
public final byte[] encrypt(byte source[])
```

Encrypt source with this key and return an array with the encrypted data. This method ultmately calls encrypt(Data,Data), but causes more memory allocation and garbage collection than using that method directly.

decrypt

```
public abstract void decrypt(Data source,
                             Data dest)
```

Decrypt the data from source with this key and put the plain data in dest. The source.buf and dest.buf arrays must be different.

decrypt

```
public byte[] decrypt(byte source[])
```

Decrypt the data from source with this key and return an array with the plain bytes. This method ultmately calls decrypt(Data,Data), but causes more memory allocation and garbage collection than using that method directly.

All Packages Class Hierarchy This Package Previous Next Index

Class IS.hi.logir.cryptonite.KeyPair

```
java.lang.Object
   |
   +----IS.hi.logir.cryptonite.KeyPair
```

public class **KeyPair**
extends Object This class is a simple holder for a pair of public/private keys. Some encryption algorithms only use a single key, in which case the public and private fields of a KeyPair may reference the same object.

Author:
Logi Ragnarsson (logir@hi.is)

See Also:
Key, KeyRing

Variable Index

- **pri**
- **pub**

Constructor Index

- **KeyPair**(Key, Key)
  Create a new KeyPair holder.

Method Index

- **getPrivate**()
  Return the private key from the pair.
- **getPublic**()
  Return the public key from the pair.

Variables

pub

```
Key pub
```

pri

```
Key pri
```

Constructors

KeyPair

```
public KeyPair(Key pub,
               Key pri)
```

Create a new KeyPair holder.

Methods

getPublic

```
public Key getPublic()
```

Return the public key from the pair.

getPrivate

```
public Key getPrivate()
```

Return the private key from the pair.

Class IS.hi.logir.cryptonite.RsaKey

```
java.lang.Object
   |
   +----IS.hi.logir.cryptonite.Key
           |
           +----IS.hi.logir.cryptonite.RsaKey
```

public final class **RsaKey**
extends Key
implements Serializable, Fingerprintable The RSA algorithm is probably the best known and most widely used public key algorithm. Breaking one RSA key is exactly as difficult as factoring the large integer that comprises the key, and there is no known way to do this in a reasonable time. Therefore RSA should be about as secure as anything if you keep your keys long. 1024 bits should be more than enough in most cases, but the truly paranoid may want to use up to 4096 bit keys.
Each RSA key is a pair (r,n) of integers and matches another key (s,n). If P is a block of plain data represented as an integer smaller than n, then it can be encrypted with the transformation:

```
E = (P^r) mod n
```

which has the inverse transformation:

```
P = (E^s) mod n
```

The key owner will keep one key secret and publish the other as widely as possible. This allows anyone who gets hold of the public key to encrypt data which can only be decrypted with the corresponding private key.
Data that is encrypted with a private key can similarly only be decrypted with the corresponding public key. This is useful for digital signatures.
When P is created from an array of bytes, it will correspond to as many bytes of plain data as the bytes needed to store n, less one.
When the encrypted integer E is converted to a byte array, it will contain as many bytes as are needed to store n. This means that the encrypted data will be slightly larger, by a ratio of (m+1)/m, with m=trunc(log2(n)/8).
Each chunk of data encrypted with RsaKey has as many bytes as the key modulo. However, the encrypted data it contains corresponds to plain data with one less byte.

Author:

Logi Ragnarsson (logir@hi.is)

See Also:

Signature

Variable Index

n
The RSA key modulo

r
The RSA key exponent

R
R is the exponent used in all public keys.

Constructor Index

RsaKey(BigInteger, BigInteger, String, String)
Create a new RSA key (r,n).

Method Index

algorithmName()
Return the name of the algorithm used by this key.
blockSize()
Returns a number n, such that least space will be wasted if a mutiple of n bytes is encrypted at once.
calcFingerprint()
Return the key's MD5 fingerprint.
createKeys(BigInteger, BigInteger, BigInteger, String, String)
Create a KeyPair object holding objects for the public RSA key (r,n) and the private RSA key (s,n).
createKeys(int, String, String)
Create a pair of public/private keys.
decrypt(Data, Data)
Decrypt the data from source with this key and put the plain data in dest.
encrypt(Data, Data)
Encrypt source with this key and put the result in dest.
equals(Object)
Return true if the two keys are eqivalent.
getSize()
Return the size of the key modulo in bits.
matches(Key)
Returns true if this and key are a matched pair of public/private keys.
toString()
Return a string representation of this key.

Variables

R

```
private static final BigInteger R
```

R is the exponent used in all public keys.

r

```
private BigInteger r
```

The RSA key exponent

n

```
private BigInteger n
```

The RSA key modulo

Constructors

RsaKey

```
public RsaKey(BigInteger r,
              BigInteger n,
              String ownerName,
              String ownerMail)
```

Create a new RSA key (r,n). It will be marked as belonging to ownerName with e-mail address ownerMail.

Methods

createKeys

```
public static KeyPair createKeys(int bitLength,
                                 String ownerName,
                                 String ownerMail)
```

Create a pair of public/private keys. The key modulo will be at most bitLength bits in length and not much shorter. It will be marked as belonging to ownerName with e-mail address ownerMail.

createKeys

```
public static KeyPair createKeys(BigInteger r,
                                 BigInteger s,
                                 BigInteger n,
                                 String ownerName,
                                 String ownerMail) throws KeyException
```

Create a KeyPair object holding objects for the public RSA key (r,n) and the private RSA key (s,n). They will be marked as belonging to ownerName with e-mail address ownerMail.

getSize

```
public int getSize()
```

Return the size of the key modulo in bits.

Overrides:

getSize in class Key

calcFingerprint

```
protected final Fingerprint calcFingerprint()
```

Return the key's MD5 fingerprint.

Overrides:

calcFingerprint in class Key

See Also:

MD5State, Fingerprintable

algorithmName

```
public String algorithmName()
```

Return the name of the algorithm used by this key.

Overrides:

algorithmName in class Key

toString

```
public String toString()
```

Return a string representation of this key.

Overrides:

toString in class Key

equals

```
public final boolean equals(Object o)
```

Return true if the two keys are eqivalent.

Overrides:

equals in class Object

matches

```
public final boolean matches(Key key)
```

Returns true if this and key are a matched pair of public/private keys.

Overrides:

matches in class Key

blockSize

```
public int blockSize()
```

Returns a number n, such that least space will be wasted if a mutiple of n bytes is encrypted at once.

Overrides:

blockSize in class Key

encrypt

```
public final void encrypt(Data source,
                          Data dest)
```

Encrypt source with this key and put the result in dest. The source.buf and dest.buf arrays must be different.

Overrides:

encrypt in class Key

decrypt

```
public void decrypt(Data source,
                    Data dest)
```

Decrypt the data from source with this key and put the plain data in dest. The source.buf and dest.buf arrays must be different.

Overrides:

decrypt in class Key