FUNDAÇÃO DE ENSINO EURÍPIDES SOARES DA ROCHA
CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

RODOLFO BARROS CHIARAMONTE

SICO: UM SISTEMA INTELIGENTE DE COMUNICAÇÃO DE DADOS COM SUPORTE DINÂMICO A SEGURANÇA

Marília
2006

RODOLFO BARROS CHIARAMONTE

SICO: UM SISTEMA INTELIGENTE DE COMUNICAÇÃO DE DADOS COM SUPORTE DINÂMICO A SEGURANÇA

Dissertação apresentada ao Programa de Mestrado em Ciência da Computação do Centro Universitário Eurípides de Marília, mantido pela Fundação de Ensino Eurípides Soares da Rocha, para obtenção do Título de Mestre em Ciência da Computação (Área de Concentração: Arquitetura de Computadores).

Orientador:
Prof. Dr. Edward David Moreno.

Marília
2006

CHIARAMONTE, Rodolfo B.
SICO: Um Sistema Inteligente de Comunicação de Dados com Suporte Dinâmico à Segurança / Rodolfo Barros Chiaramonte; orientador: Prof. Dr. Edward David Moreno. Marília, SP: [s.n.], 2006.
111 f.
Dissertação (Mestrado em Ciência da Computação) – Centro Universitário Eurípides de Marília – Fundação de Ensino Eurípides Soares da Rocha.

1. Arquitetura de Sistemas 2. Segurança 3. Criptografia

CDD: 005.82

RODOLFO BARROS CHIARAMONTE

SICO: UM SISTEMA INTELIGENTE DE COMUNICAÇÃO DE DADOS COM SUPORTE DINÂMICO A SEGURANÇA

Banca examinadora da dissertação apresentada ao Programa de Mestrado em Ciência da Computação da UNIVEM, /F.E.E.S.R, para obtenção do Título de Mestre em Ciência da Computação. Área de concentração: Arquitetura de Computadores.

Resultado: _____

ORIENTADOR: Prof. Dr. _____

1º EXAMINADOR: _____

2º EXAMINADOR: _____

Marília, 10 de Abril de 2006.

Dedico aos meus pais, Alcides e Marta, e à minha irmã, Josiane, e todos meus familiares.

AGRADECIMENTOS

Agradeço a Deus pois sem Ele nada seria possível.

Agradeço ao Prof. Dr. Edward por ter me apoiado e incentivado por todos estes anos.

Agradeço a meus pais, Alcides e Marta, e à minha irmã, Josiane, pelo estímulo, paciência e compreensão em todos os momentos. Agradeço também a todos os meus familiares.

Agradeço aos amigos e colegas de laboratório, em especial: Ana Paula, César, Fábio, Fernando, Karina, Larissa, Marcos Piva, Paulo, Reinaldo pelas sugestões e apoio.

Enfim, agradeço a todos que direta ou indiretamente contribuíram para a conclusão deste trabalho.

Você vê coisas e diz: Por que?; mas eu sonho coisas que nunca existiram e digo: Por que não? (George Bernard Shaw)

CHIARAMONTE, Rodolfo. **SICO: Um Sistema Inteligente de Comunicação de Dados com Suporte Dinâmico a Segurança**. 2006. 111 f. Dissertação (Mestrado em Ciência da Computação) – Centro Universitário Eurípides de Marília – Fundação de Ensino Eurípides Soares da Rocha, Marília, 2006.

RESUMO

Atualmente existe uma grande quantidade de algoritmos criptográficos, o que dificulta muito definir qual será o melhor algoritmo a ser utilizado em uma determinada conexão. O projeto tem como objetivo propor e implementar um sistema que consiga definir através de alguns parâmetros coletados do ambiente em que estiver sendo executado qual é o algoritmo ou conjunto de algoritmos mais indicado para uma determinada conexão segura. Inicialmente foram implementados diversos algoritmos de criptografia, e posteriormente, foi proposto o sistema para o controle da comunicação, bem como um protocolo que permita alternar, dinamicamente, as chaves e os algoritmos. A implementação do sistema, chamado de SICO – Sistema Inteligente de Comunicação, foi desenvolvida utilizando a linguagem Java. Foram então realizados testes com uma aplicação para a transferência de um arquivo via rede onde foi possível perceber que o sistema atendeu satisfatoriamente os requisitos para o qual foi projetado.

Palavras - chave: segurança, criptografia, qualidade de serviço, troca de chaves.

CHIARAMONTE, Rodolfo. **SICO: Um Sistema Inteligente de Comunicação de Dados com Suporte Dinâmico a Segurança**. 2006. 111 f. Dissertação (Mestrado em Ciência da Computação) – Centro Universitário Eurípides de Marília – Fundação de Ensino Eurípides Soares da Rocha, Marília, 2006.

ABSTRACT

Currently a great amount of cryptography algorithms exists, it makes difficult to define which will be the best algorithm to be used in one determined connection. The project has as objective to implement a system able to define, through some collected parameters of the environment where it will be being executed. Initially, many cryptography algorithms was implemented, and later, the system for the control of the communication as well as a protocol that allows alternating the keys and the algorithms dynamically was implemented. The implementations was being developed using the Java language and its called SICO - Sistema Inteligente de Comunicação. It was tested an application for file transference using a network, and it was possible realize that the system satisfactorily accomplished the system goals.

Keywords: Security, Cryptography, Quality of Service, Key Exchange

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS

3DES – *Triple DES*

AES – Advanced Encryption Standard

ANSI – *American National Standards Institute*

API – *Application Programming Interface*

ARPANET – *Nome da rede criada pelo DARPA*

ASCII – *American Standard Code for Information Interchange*

CBR – *Constant Bit Rate*

CPU – *Central Processing Unit*

DARPA – *U.S. Department of Defense Advanced Research Projects Agency*

DES – *Data Encryption Standard*

EFF – *Electronic Frontier Foundation*

IDEA – *International Data Encryption Algorithm*

IP – *Internet Protocol*

IPES – *Improved PES*

JAES – *AES implementado através de JNI*

JCA – *Java Cryptography Architecture*

JNI – *Java Native Interface*

LAN – *Local Area Network*

MAC – *Message Authentication Code*

MAN – *Metropolitan Area Network*

MD4 - *Message Digest 4*

MD5 – *Message Digest 5*

MIT – *Massachsetts Institute of Technology*

NBS – *National Bureau of Standards*

NIST – *National Institute of Standards and Technology*

NSA – *National Security Agency*

PKI – *Public Key Infraestructure*

PLC – *Power Line Communication*

QoS – *Quality of Service*

RAM – *Random Access Memory*

RC6 – *Rivest Cipher 6*

RSA – *Rivest, Shamir e Adleman*

SHA-1 – *Secure Hash Algorithm*

SHS – *Secure Hash Standard*

SICO – *Sistema Inteligente de Comunicação*

SSL – *Secure Socket Layer*

TCP – *Transmission Control Protocol*

USC – *University of Southern California*

VBR – *Variable Bit Rate*

VoD – *Video on Demand*

VoIP – *Voice over IP*

WAN – *Wide Area Network*

WEP – *Wireless Encryption Protocol*

SUMÁRIO

CAPÍTULO 1 – DESCRIÇÃO GERAL DO PROJETO

Hoje em dia, devido ao grande crescimento das redes de computadores e das aplicações que as utilizam, existe um grande número de algoritmos de criptografia e sistemas de segurança que surgiram com a necessidade de preservar as informações que trafegam por essas redes.

Esses algoritmos se diferem quanto ao objetivo (confidencialidade, integridade, autenticação, etc.), nível de segurança e desempenho; com isso, quando uma conexão segura tiver de ser estabelecida, há a necessidade de escolher o conjunto de algoritmos que deverá ser responsável pela segurança das informações. Essa escolha não é simples, pois como descrito anteriormente, existe um grande número deles, cada um com suas características e objetivos. Além de definir o algoritmo, existe também a necessidade de definir uma chave ou conjunto de chaves para essa comunicação, o que também impõe a necessidade de escolher a configuração necessária para elas (tamanho em bits e outros parâmetros utilizados na escolha de chaves).

Tendo em vista a complexidade em escolher o conjunto algoritmos/chaves para a conexão segura a ser estabelecida, foi proposto um sistema que avalie as características necessárias para a comunicação, tais como: o desempenho, o nível de segurança desejado, a velocidade da rede utilizada, o tempo de processamento disponível nas estações e outros parâmetros; e com base nesses parâmetros e nas características presentes nos algoritmos seja apresentado um conjunto que melhor satisfaça as condições estabelecidas pelo ambiente para a conexão segura desejada.

Outro fator importante para esse sistema é manter a possibilidade de alternar os algoritmos e as chaves dinamicamente, já que vários parâmetros que influenciam na escolha

são dinâmicos e o conjunto escolhido no início da conexão pode não ser o mais indicado em outros instantes da mesma.

Os sistemas que existem atualmente permitem a escolha do algoritmo e da chave no início da conexão e utilizam este conjunto enquanto ela existir. O protocolo SSL (*Secure Socket Layer*) (FREIER et. al., 2004) (DIERKS e ALLEN, 1999) por exemplo, define o conjunto algoritmo/chave somente no início da conexão; outro fator importante é que esta escolha não considera alguns parâmetros importantes que afetam o desempenho, como por exemplo, o tempo de CPU (*Central Processing Unit*) necessário para executar um determinado algoritmo criptográfico em conjunto com a execução da aplicação.

Um outro exemplo de sistema que realiza a escolha do conjunto algoritmo/chave somente no início da conexão é a *middleware* CriptoQoS apresentada por Meylan (2003). No entanto esta escolha considera alguns parâmetros importantes (utilização de CPU, utilização de memória, utilização de interface de rede, utilização de disco), inclusive parâmetros de QoS (Quality of Service – Qualidade de Serviço).

1.1 – Justificativa

A Figura 1 mostra o gráfico de desempenho de vários algoritmos criptográficos, com ênfase nos finalistas do concurso AES. Através dele é possível perceber que a escolha do algoritmo e do tamanho da chave tem grande influência sobre o desempenho. Apesar do algoritmo RSA (Rivest, Shamir e Adleman) não estar no gráfico é importante considerar que utiliza-lo para manter uma comunicação que necessita somente de confidencialidade pode não ser interessante pois é um algoritmo assimétrico que objetiva não só a confidencialidade como

também a autenticação, o que o torna mais lento que outros algoritmos, como por exemplo os algoritmos DES (*Data Encryption Standard*) e AES (*Advanced Encryption Standard*) que são algoritmos simétricos.

Portanto, pode-se notar a necessidade e a importância da escolha correta do algoritmo a ser utilizado em uma conexão. Importante destacar que na Figura 1 foram apresentados dados de desempenho de vários algoritmos criptográficos que são apresentados e discutidos posteriormente. Estes dados consideram o uso dos algoritmos criptográficos para cifrar um arquivo de vídeo de tamanho fixo. Importante notar que há uma grande variação de desempenho entre os algoritmos apresentados. Por exemplo, pode-se perceber que o algoritmo RC6 (*Rivest Cipher 6*) possui um desempenho duas vezes melhor que o Serpent.

Com isso, o projeto tem como objetivo implementar um sistema de comunicação seguro que atenda a critérios preestabelecidos para a escolha de um algoritmo que melhor se adeque às condições necessárias para a conexão a ser estabelecida; esse sistema também deverá detectar dinamicamente a alteração de parâmetros importantes para a escolha e definir a necessidade de trocar o algoritmo ou a chave dinamicamente durante a conexão em função dos parâmetros de rede, aplicação e Qualidade de Serviço.

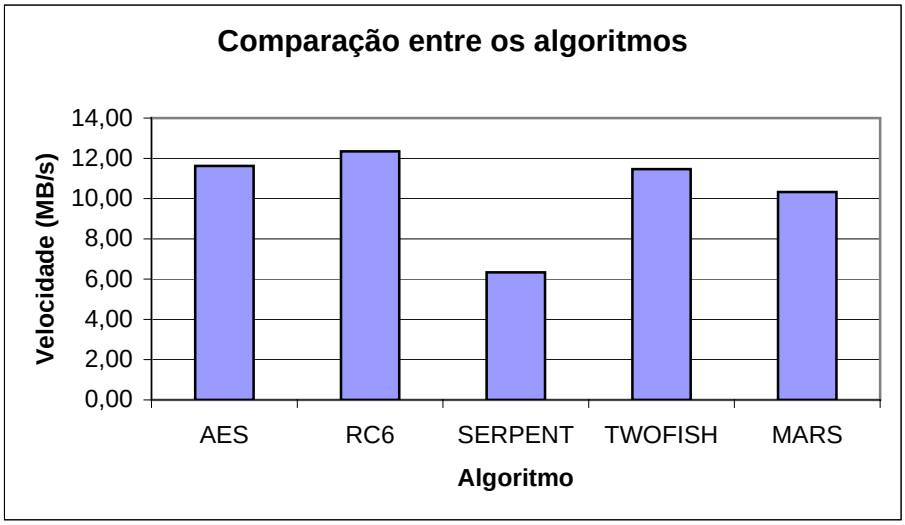

Figura 1 – Desempenho de Vários Algoritmos Criptográficos

1.2 – Objetivos da Dissertação

O projeto teve como objetivo implementar um sistema com diversos algoritmos criptográficos, capaz de escolher através de alguns parâmetros coletados do ambiente o melhor algoritmo ou conjunto de algoritmos necessário para uma determinada comunicação. Além disto, o sistema deverá ser capaz de detectar alterações nos parâmetros que influenciam na escolha do conjunto algoritmo/chave e, caso necessário, realizar a troca da chave dinamicamente.

Durante o projeto foram implementados os algoritmos criptográficos: Posicional, DES, AES, IDEA (*International Data Encryption Standard*), RSA, MD5 (*Message-Digest* 5), SHA-1 (*Secure Hash Algorithm*). Inicialmente estes algoritmos foram implementados em linguagem C. Posteriormente, esta implementação em C serviu de base para implementações utilizando a linguagem Java. Foram também realizados testes de desempenho utilizando outras implementações, a saber: implementações dos finalistas ao AES pelo Dr. Brian Gladman (GLADMAN, 2006) e implementações dos algoritmos DES e AES presentes na JCA (*Java Cryptography Architecture*). Estes testes auxiliaram na escolha das implementações a se utilizar no sistema.

Por sua vez, também foi implementado o sistema para o controle da comunicação, bem como um protocolo que permite alternar as chaves e algoritmos dinamicamente.

Desta forma, outros objetivos foram relacionados para se chegar ao objetivo principal, eles podem ser resumidos em:

1 – Estudar os vários algoritmos criptográficos levantando dados sobre seu desempenho, características e nível de segurança;

2 – Implementar esses algoritmos em linguagem C e Java;

3 – Implementar um sistema simples de comunicação de dados;

4 – Implementar um sistema que consiga captar alguns parâmetros do ambiente em que está sendo executado (velocidade da rede, carga de processamento, etc.);

5 – Implementar um sistema final que permita a comunicação segura de dados e através dos parâmetros captados do ambiente consiga definir um melhor algoritmo para ser utilizado e trocá-lo dinamicamente caso necessário. Este sistema é chamado de SICO (Sistema Inteligente de Comunicação).

1.3 – Metodologia

Inicialmente foi realizado um estudo sobre os algoritmos de criptografia citados anteriormente e realizadas implementações em linguagem C e posteriormente Java. Após isso, foram realizados testes de desempenho para avaliar a velocidade dos algoritmos. Foram também avaliadas outras implementações para verificar em qual era possível observar um melhor desempenho.

Após essa fase foi implementado um sistema que recebe conexões de uma rede, detecta os parâmetros necessários para definir o algoritmo/chave a ser utilizado e envia os dados cifrados pela rede. Também foi implementado um sistema que receba os dados cifrados e os decifrem no computador destino.

Na etapa de testes e avaliação dos resultados avaliou-se o desempenho do sistema final. Com isto, para a avaliação do desempenho, criou-se uma aplicação sobre o sistema para detectar medidas como o *overhead* causado pelo sistema e a taxa de transmissão média.

1.4 – Organização da Dissertação

Esta dissertação está organizada em seis capítulos, sendo que no Capítulo 2 são apresentados conceitos básicos de criptografia bem como alguns algoritmos que são utilizados na implementação do sistema. No Capítulo 3 são discutidos alguns conceitos sobre redes, QoS e segurança enfatizando alguns sistemas de QoS e a segurança provida por estes sistemas. A estrutura inicial do sistema proposto é então apresentada no Capítulo 4. O Capítulo 5 descreve a implementação do sistema, bem como alguns resultados alcançados com a utilização do mesmo. As conclusões e os trabalhos futuros são finalmente apresentados no Capítulo 6.

CAPÍTULO 2 – CONCEITOS DE CRIPTOGRAFIA

A criptografia é utilizada desde a época da escrita hieroglífica dos egípcios e não mudou muito até meados do século XX. Durante a Segunda Guerra Mundial algumas nações investiram na criptografia para a transmissão de informações secretas. Foi a partir desse período que, com a invenção dos computadores, ocorreu uma grande mudança nos métodos de criptografia uma vez que centenas de cálculos podiam ser realizados em questão de segundos. Esse novo modelo de criptografia tem base na utilização de algoritmos matemáticos complexos.

Até a década de 70 todos os algoritmos eram secretos, principalmente os utilizados pelas forças armadas. O primeiro algoritmo de criptografia cujo conhecimento se tornou público foi o DES que foi desenvolvido pela IBM em meados da década de 70 em um projeto liderado por Walter Tuchman e Carl Meyer (TERADA, 2000). O algoritmo DES (MATSUI, 1993) foi então publicado no *National Bureau of Standards* (NBS) em 1977 para ser adotado como padrão nos EUA. A partir desse ano tornou-se efetivamente um padrão.

Em 1976, W. Diffie e M. Hellman publicaram um artigo seminal que inspirou os algoritmos de chave assimétrica (aqueles que possuem chave pública e secreta em que geralmente a chave pública é utilizada para cifrar e a secreta para decifrar). Nesse artigo (DIFFIE e HELLMAN, 1976) os autores se basearam no Problema do Logaritmo Discreto o que torna computacionalmente difícil calcular a chave secreta a partir da chave pública.

2.1 – Tipos e Objetivos

Com a criação da Internet surgiram várias facilidades eletrônicas tais como o comércio eletrônico, bancos eletrônicos e outros métodos que exigem a transmissão de informações confidenciais através de redes consideradas inseguras. Uma das soluções para este problema é a criptografia em que uma informação secreta é criptografada utilizando uma chave e somente quem a conhece consegue ler essas informações. Este objetivo de ocultar uma informação para que somente as pessoas autorizadas possuam acesso chama-se Confidencialidade.

Mas a Confidencialidade não é o único objetivo que pode ser alcançado com o auxílio da criptografia. Segundo Stallings (1998), existem outros serviços de segurança que podem ser alcançados, tais como:

- Confidencialidade: Garante que a informação transmitida ou armazenada seja acessível apenas pelas entidades autorizadas;
- Autenticação: Garante que o emissor de uma mensagem não seja falso;
- Integridade: Garante que uma mensagem não seja alterada por entidades não autorizadas durante o armazenamento ou trânsito.
- Não Repúdio: Garante que o emissor ou receptor de uma mensagem não seja capaz de negar sua transmissão ou recepção;
- Controle de Acesso: Garante que uma informação apenas possa ser manipulada por entidades autorizadas.
- Disponibilidade: Garante que um recurso sempre esteja disponível para entidades autorizadas.

Existem dois tipos de algoritmos criptográficos: Os algoritmos de chave simétrica e os algoritmos de chave assimétrica.

Os algoritmos de **chave simétrica** possuem apenas uma chave secreta para cifrar ou decifrar a mensagem. Isso pode gerar dificuldades para o envio da chave secreta já que é necessário um meio seguro para transportá-la. Um exemplo de algoritmo que se baseia no princípio de chave simétrica é o algoritmo DES (SCHNEIER, 1996). Por exemplo, imagine que uma informação secreta será enviada por uma longa distância, mas a chave tem que ser modificada pois alguém pode ter descoberto a chave antiga. Retransmitir uma nova chave de modo seguro é um problema. A solução para isso é a utilização de algoritmos de chave assimétrica.

Os algoritmos de **chave assimétrica** possuem duas chaves, uma para cifrar (chamada chave pública) e outra para decifrar (chamada chave privada) a mensagem. Se uma mensagem for cifrada com a chave pública, somente a chave privada pode decifrá-la; caso a mensagem for cifrada com a chave privada, somente a chave pública poderá decifrá-la. Com esse algoritmo é computacionalmente difícil calcular a chave secreta a partir do conhecimento da chave pública. Um algoritmo baseado no princípio de chave assimétrica muito conhecido é o algoritmo RSA (RIVEST et al, 1978).

Os **sistemas híbridos** consistem na união dos sistemas simétrico e assimétrico. Neste tipo de sistema, o algoritmo de chave assimétrica é utilizado para transmitir a chave de um algoritmo simétrico para ser utilizado na conexão; e os algoritmos simétricos são os mais utilizados para o envio de mensagens criptografadas.

2.2 – Segurança de um algoritmo de Criptografia

A segurança de um algoritmo está ligada à dificuldade computacional de se obter o texto legível a partir de um texto ilegível, ou de se obter a chave a partir de conjuntos de textos legíveis com seus respectivos ilegíveis.

Desta forma, se é possível obter o texto legível a partir de um texto ilegível utilizando um método computacionalmente fácil, pode-se dizer que o texto ilegível é quebrável; por outro lado, se é possível obter a chave a partir de um conjunto com um número polinomial (ao comprimento da chave) de textos legíveis com seus respectivos ilegíveis, pode-se dizer que a chave ou que o algoritmo é quebrável.

Caso a chave for quebrável, qualquer informação ilegível criada com esta chave também é quebrável.

Não se conhece um método matemático capaz de definir se um algoritmo criptográfico é seguro ou não. Com isso, para se comprovar a segurança de um algoritmo, é necessário publicá-lo para que outros pesquisadores possam tentar atacá-lo com os métodos mais sofisticados. Se o algoritmo suportar os ataques, pode-se dizer que este algoritmo é seguro.

Existem vários tipos de ataque possíveis para um algoritmo, a seguir são descritos alguns tipos de ataque mais comuns (TERADA, 2000):

- Ataque por só-texto-ilegível: este ataque tenta obter o texto legível a partir somente de um texto ilegível. Se este tipo de ataque for viável, o algoritmo é considerado totalmente inseguro.

- Ataque por texto legível conhecido: este ataque tenta obter a chave a partir de conjuntos de textos legíveis com seus respectivos ilegíveis.

- Ataque por texto legível escolhido: neste ataque é possível escolher um conjunto de textos legíveis com seus respectivos ilegíveis. Esta escolha deve ser feita analisando alguma característica estrutural que permita obter maior conhecimento sobre o algoritmo e a chave.

2.3 – Algoritmos Básicos de Criptografia

Nesta seção são apresentados alguns algoritmos básicos de criptografia, que são conhecidos por sua simplicidade e importância histórica. Adicionalmente se apresenta o algoritmo Posicional, que é um algoritmo didático.

2.3.1 – Cifra de César

O primeiro algoritmo discutido é a Cifra de César que consiste em substituir uma letra do alfabeto por outra de acordo com um deslocamento circular, assim a chave define quanto a letra será deslocada.

Por exemplo, observando a linha original da Tabela 1 e supondo uma chave de valor 9, a linha 3 dessa será composta deslocando cada letra da linha original em 9 posições. Assim, a letra A que está na posição 1 deslocada em 9 equivale à letra J que está na posição 10 na segunda linha da tabela e é colocada na posição 1 da terceira linha, do mesmo modo, a letra B (posição 2) é substituída por K (posição 11 na segunda linha), e assim as outras letras são calculadas.

Tabela 1 – Deslocamento utilizando a cifra de César

1	2	3	4	5	6	7	8	9	10	11	12	13
A	B	C	D	E	F	G	H	I	J	K	L	M
J	K	L	M	N	O	P	Q	R	S	T	U	V

14	15	16	17	18	19	20	21	22	23	24	25	26
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
W	X	Y	Z	A	B	C	D	E	F	G	H	I

Dessa maneira supondo que se tem a palavra original TESTE, ela ficaria criptografada como CNBCN.

Nesse algoritmo são possíveis apenas 25 chaves diferentes já que o número de letras utilizadas é 26 e o número máximo de deslocamentos sem repetições é 25. Nesse caso é fácil testar todas as chaves possíveis para obter a mensagem original a partir da mensagem cifrada. Com isso seria possível descobrir a chave rapidamente até sem o auxílio do computador.

2.3.2 – Algoritmo de Substituição Simples

O algoritmo de Substituição Simples consiste em substituir uma letra por outra qualquer definida por uma chave. Esse algoritmo possui muito mais chaves possíveis, portanto é mais difícil calcular a mensagem original através do teste de todas as chaves. Com isso o nível de segurança oferecido por esse algoritmo é maior que o oferecido pelo algoritmo anterior, pois um dos fatores que influi no nível de segurança é o número de chaves possíveis sem repetição.

O funcionamento desse algoritmo é simples: São escolhidas várias letras aleatoriamente, sem repetição, para ser a chave do algoritmo.

No exemplo da Tabela 2, a segunda linha é a chave, que foi escolhida conforme critério especificado anteriormente. Por exemplo, para cifrar a letra A foi gerada outra letra escolhida aleatoriamente, no nosso caso a letra Q, para cifrar a letra B foi gerada outra letra aleatoriamente com exceção do Q que já foi utilizado anteriormente, no nosso caso a letra gerada foi R, as outras letras são selecionadas do mesmo modo.

Tabela 2 – Chave da Substituição Simples

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
Q	R	P	T	W	M	V	X	N	Z	C	F	J	L	O	K	E	S	A	Y	B	U	H	I	G	D

Para cifrar uma determinada palavra com base na Tabela 2 substitui-se as letras originais (1ª linha) pelas da chave (2ª linha), dessa maneira supondo que se tem a palavra original TESTE, ela ficaria criptografada como YWAYW.

Com o critério utilizado para a formação da chave, o número de chaves possíveis nesse algoritmo é a permutação das 26 letras, o que equivale a 26! ou $4{,}03291461126605635584 \times 10^{26}$. Com esse algoritmo o tempo gasto para testar todas as chaves possíveis é muito grande o que o torna mais seguro que o anterior (Cifra de César).

No entanto esse algoritmo ainda possui um ponto fraco, pois ele ainda preserva a freqüência das letras. A palavra cifrada (YWAYW) possui dois Y, dois W e um A. A palavra original (TESTE) possui dois T, dois E e um S. Verifica-se portanto que a freqüência das letras são mantidas, isso também ocorre no primeiro algoritmo discutido.

Portanto os algoritmos de Cifra de César e Substituição Simples possuem um grave problema de segurança, esses algoritmos preservam a freqüência das letras o que pode facilitar um ataque a partir do texto cifrado, sem a necessidade de testar todas as chaves possíveis.

2.3.3 – Cifra de Vigenère

O algoritmo da Cifra de Vigenère não possui o problema de manter a freqüência das letras. Esse algoritmo consiste em cifrar as mensagens somando a mensagem original com a chave, por exemplo: Imagine que o valor para a letra A, B, C, D, ..., Z são respectivamente 1, 2, 3, 4, ..., 26, assim para cifrar a palavra TESTE utilizando a chave AB, escolhida aleatoriamente, o texto cifrado será UGTVF (veja Tabela 3 e Tabela 4).

Tabela 3 – Valores dos Caracteres

	A	B	C	D	E	F	G	H	I	J	K	L	M
00	01	02	03	04	05	06	07	08	09	10	11	12	13

N	O	P	Q	R	S	T	U	V	W	X	Y	Z
14	15	16	17	18	19	20	21	22	23	24	25	26

Tabela 4 – Exemplo da Cifra de Vigenère

Mensagem Original	T	E	S	T	E
Valor de cada letra	20	5	19	20	5
Chave repetida	A	B	A	B	A
Valor de cada letra da chave	1	2	1	2	1
Soma dos valores	21	7	20	22	6
Mensagem Cifrada	U	G	T	V	F

Note que nesse caso a letra T e a letra E aparecem 2 vezes no texto original e, no texto cifrado nenhuma letra aparece mais que uma vez. Isso significa que a freqüência das letras não é preservada.

No entanto, esse algoritmo ainda não é muito seguro. Existem meios de criptoanálise simples que conseguem quebrar este algoritmo rapidamente (TERADA, 2000), como por exemplo, um ataque por texto legível conhecido que foi descrito na seção 2.2.

2.3.4 – Um algoritmo Didático (Posicional)

O algoritmo de criptografia posicional consiste em que a posição do byte interfere sobre a chave utilizada na criptografia. O algoritmo Posicional se baseia no bem conhecido algoritmo Cifra de César. Para formar o Algoritmo Posicional foi introduzida uma função sobre o algoritmo Cifra de César para que cada posição seja criptografada de forma diferente eliminando o problema de freqüência de letras.

Com o algoritmo Posicional Básico a seqüência AAABBB, por exemplo, poderia ser criptografada como BCDFGH, sem acrescer a ela nenhum bit. Para isso, admita-se o valor decimal dos bytes A, B, C, D, E, F, G e H de acordo com a tabela ASCII (*American Standard Code for Information Interchange*) (65, 66, 67, 68, 69, 70, 71 e 72 respectivamente). A esses valores são acrescidos os valores de sua posição. Desta forma, supõe-se o exemplo da Tabela 5 onde existe uma seqüência AAABBB. Como o primeiro A está na posição um, o seu valor decimal criptografado será 66 (que equivale na tabela ASCII ao caractere B), mas como o segundo A está na posição dois, deve-se acrescer 2 ao seu valor ASCII, assim o valor decimal do segundo byte será 67 (que equivale na tabela ASCII ao caractere C).

Tabela 5 – Exemplo do Algoritmo Posicional Básico

Seqüência	A	A	A	B	B	B
Valor decimal na tabela ASCII	65	65	65	66	66	66
Valor da posição	1	2	3	4	5	6
Valor decimal ASCII (código criptografado)	66	67	68	70	71	72
Código em caracteres (criptografado)	B	C	D	F	G	H

Esse modelo ainda é simples de ser quebrado. Por esse motivo é recomendável somar ao valor do byte original representado em ASCII um número gerado utilizando-se de uma

expressão matemática que usa a posição do byte (variável x) (CHIARAMONTE e MORENO, 2002). Por exemplo: usando uma expressão do tipo a*x+b.

Nesse tipo de criptografia, quanto maior o grau da expressão posicional, maior será a complexidade do algoritmo e por sua vez a segurança dos dados cifrados usando-se desse algoritmo. Por exemplo: Se a expressão posicional for uma equação do terceiro grau, terá uma segurança maior que um sistema com equação do segundo grau.

A seguir é apresentado um exemplo completo de criptografia utilizando grau 3. Neste exemplo ocorre o estouro de um byte, ou seja, o valor decimal do código criptografado é maior que 256 que é o valor máximo que pode ser armazenado em um byte (o código ASCII tem a capacidade de 2^8=256 possibilidades). No caso de estouro, assume-se que o valor do byte será "valor encontrado mod 256" sendo que mod retorna o resto da divisão.

A equação posicional utilizada será: $f(x) = 23x^3 + 26x^2 – 45x^1 – 63$. (Os coeficientes (23,26,-45,-63) foram selecionados aleatoriamente.) Considerando o mesmo exemplo anterior cuja seqüência original é AAABBB, temos as seguintes fases (ver Tabela 6).

Tabela 6 – Exemplo Posicional de Grau 3

Seqüência	A	A	A	B	B	B
Valor decimal na tabela ASCII	65	65	65	66	66	66
Valor da posição (x)	1	2	3	4	5	6
Resultado da expressão: $23x^3 + 26x^2 - 45x – 63$	-59	135	657	1645	3237	5571
Valor da expressão + valor ASCII original	6	200	722	1711	3303	5637
Valor decimal na tabela (código criptografado)	6	200	210	175	231	5
Código em caracteres (criptografado)		+	Ê	»	Þ	

Na Tabela 6, as colunas em destaque com sombra indicam aquelas onde ocorreu o estouro de um byte. Os valores 722, 1711, 3303 e 5637 ultrapassam 256 e não podem ser armazenados em apenas um byte. Para solucionar esse problema é realizada uma divisão por

256 e o resto da divisão é utilizado como o valor criptografado. Por exemplo: dividindo o valor 722 por 256, o resto obtido é 210 que é utilizado na penúltima linha da tabela. Os outros valores são calculados do mesmo modo.

Por questões de otimização, para a implementação desta operação pode ser utilizada a operação AND entre o valor desejado e 255.

Interessante destacar que também foram desenvolvidas outras versões otimizadas do algoritmo Posicional, chamadas de: Posicional 32 bits; e Posicional 32 bits com bits aleatórios. Ver detalhes em (CHIARAMONTE et. al., 2005).

2.4 – Outros Algoritmos: Desempenho e segurança

Nesta seção são apresentados quatro algoritmos, sendo os três primeiros algoritmos simétricos e o último assimétrico. Desta forma, é possível verificar a diferença de desempenho entre tipos diferentes de algoritmos.

2.4.1 – Algoritmo Simétrico IDEA

O algoritmo IDEA, inicialmente chamado de IPES (*Improved PES*), foi proposto por X. Lai e J. Massey em 1991 (LAI e MASSEY, 1991). Foi projetado para ser eficiente em implementações por software (TERADA, 2000).

O IDEA possui chave secreta de 128 bits e tanto a entrada (texto legível) quanto a saída (texto ilegível) são de 64 bits. O mesmo algoritmo serve tanto para cifrar quanto para decifrar e consiste de 8 iterações utilizando sub-chaves distintas e uma transformação final.

As operações utilizadas pelo IDEA são todas sobre 16 bits conforme descrito a seguir:

⊕ : Ou exclusivo (XOR) sobre 16 bits;

+ : Soma mod 2^{16}, ou seja, somar dois valores de 16 bits desprezando o mais à esquerda, correspondente a 2^{16};

⊙ : Esta operação consiste em vários passos:

1. Multiplicar dois valores de 16 bits obtendo um novo valor, chamado de Z, sendo que antes de multiplicar, se um desses valores for 0 deve ser alterado para 2^{16}.
2. Calcular Z mod (2^{16} + 1), ou seja, o resto da divisão de Z por 2^{16} + 1.
3. Se o resultado da operação acima for 2^{16}, então o resultado final da operação é 0, caso contrário é o valor obtido em 2 (Z mod (2^{16} + 1)).

Como o mesmo algoritmo é utilizado tanto na cifragem quanto na decifragem, o que define qual a operação a ser realizada é a geração das sub-chaves.

A Tabela 7 apresenta uma representação da criação das 52 sub-chaves utilizadas para a criptografia, sendo cada sub-chave de 16 bits, formadas a partir da chave secreta de 128 bits (K). A primeira sub-chave (K1) é gerada considerando os 16 bits mais significativos de K, a segunda (K2) é gerada considerando os próximos 16 bits, e assim são geradas as sub-chaves até K8 que é formada pelos 16 bits menos significativos de K. Como a cada 8 sub-chaves geradas a próxima sub-chave inicia-se com um deslocamento de 25 bits a partir do início da chave K a 9ª sub-chave é formada por 16 bits a partir do 25º bit a partir da direita de K, a K10 é formada pelos 16 próximos bits, e assim até gerar K14. A 15ª sub-chave (K15), é formada

pelos 7 bits menos significativos de K e para completá-la são usados os primeiros 9 bits (os 9 bits mais significativos de K).

Lembrando que a cada 8 sub-chaves a próxima chave inicia-se acrescentando um deslocamento de 25 bits a K, a K17 inicia-se a partir do 50º bit a partir da direita de K. Dessa forma são geradas as primeiras 48 sub-chaves.

A sub-chave K49 inicia-se a partir do 22º bit a partir da direita de K, a K50 é formada pelos próximos 16 bits, e assim até formar a K52.

Tabela 7 – Criação das Sub-chaves no algoritmo IDEA

	Bit de início em K (a partir da direita)							
K1..K8	0	16	32	48	64	80	96	112
K9..K16	25	41	57	73	89	105	121	9
K17..K24	50	66	82	98	114	2	18	34
K25..K32	75	91	107	123	11	27	43	59
K33..K40	100	116	4	20	36	52	68	84
K41..K48	125	13	29	45	61	77	93	109
K49..K52	22	38	54	70				

Como descrito anteriormente, o algoritmo IDEA possui 8 iterações e uma transformação final. Cada iteração possui duas partes e utiliza 6 sub-chaves que são chamadas de Ka, Kb, Kc, Kd, Ke e Kf.

A primeira parte de cada iteração usa as quatro primeiras sub-chaves (Ka, Kb, Kc, Kd); e a segunda parte de cada iteração usa as sub-chaves Ke e Kf; e utilizam as operações descritas no início. As duas partes utilizam tanto entrada quanto saída de 64 bits dividida em quatro blocos de 16 bits que são chamadas de Xa, Xb, Xc e Xd (para a entrada) e Xa’, Xb’, Xc’ e Xd’ (para a saída); e a saída da primeira parte é a entrada para a segunda parte.

O algoritmo para a primeira parte da iteração é:

```
Xa' = Xa ⊙ Ka
Xd' = Xd ⊙ Kd
Xb' = Xc + Kc
Xc' = Xb + Kb
```

O algoritmo para a segunda parte da iteração é:

```
Y1 = Xa ⊕ Xb
Z1 = Xc ⊕ Xd
Y2 = [(Ke ⊙ Y1) + Z1] ⊙ Kf
Z2 = (Ke ⊙ Y1) + Y2
Xa' = Xa ⊕ Y2
Xb' = Xb ⊕  Y2
Xc' = Xc ⊕ Z2
Xd' = Xd ⊕ Z2
```

A ultima transformação utiliza as últimas quatro sub-chaves (K49, K50, K51 e K52). Após 8 iterações descritas anteriormente, a saída Xa', Xb', Xc' e Xd' é fornecida como entrada da transformação final gerando o texto criptografado Xa'', Xb'', Xc'' e Xd'' como descrito a seguir.

```
Xa'' = Xa' ⊙ K49
Xd'' = Xd' ⊙ K52
Xb'' = Xc' + K50
Xc'' = Xb' + K51
```

Para decifrar uma mensagem com o algoritmo IDEA basta alterar a forma de como as sub-chaves são geradas. É necessário calcular a inversa das chaves utilizadas na primeira parte de cada iteração e inverter a ordem em que elas são utilizadas.

Inicialmente são utilizadas as inversas multiplicativas mod $2^{16} + 1$ das sub-chaves K49 e K52 para formar as sub-chaves K1 e K4 e as inversas aditivas das sub-chaves K50 e K51 para gerar as sub-chaves K2 e K3.

Utilizar as sub-chaves 47 e 48 para formar K5 e K6 que são utilizadas na segunda parte da primeira iteração.

Utilizar as sub-chaves inversas das chaves K43, K44, K45 e K46 na primeira parte da segunda iteração e as sub-chaves K41 e K42 na segunda parte da segunda iteração. E assim por diante até utilizar K1, K2, K3 e K4 na transformação final.

Uma versão do algoritmo IDEA foi implementada em linguagem C. Através desta implementação pode-se perceber que o algoritmo IDEA possui um bom desempenho chegando a atingir taxas de cifragem de até 500Kbytes/segundo em um computador Duron 950 Mhz com 128Mb de memória. O gráfico da Figura 2 mostra uma comparação das duas versões (utilizando Blakley como multiplicação modular, e utilizando o algoritmo Low-High para a multiplicação (mod $2^{16} + 1$) e qual o impacto gerado pelo uso dessas implementações).

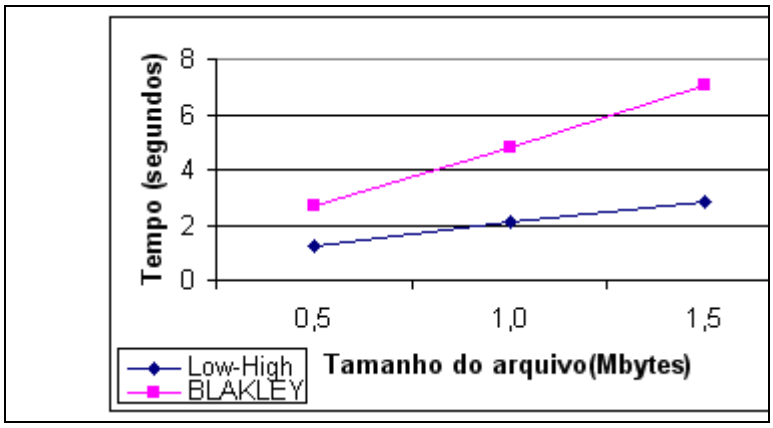

Figura 2 – Desempenho do Algoritmo IDEA (CHIARAMONTE, 2003)

2.4.2 – Algoritmo Simétrico DES

O algoritmo DES é um dos algoritmos de cifragem mais usado no mundo. Durante muitos anos, e para muitas pessoas, criptografia e DES foram sinônimos. Apesar do recente grande feito da *Electronic Frontier Foundation* (EFF), criando uma máquina de US$ 220.000

para quebrar mensagens cifradas com DES, este algoritmo vai continuar sendo útil em governos e bancos pelos próximos anos através da versão chamada triple-DES (TKOTZ, 2003)(PEREIRA e MORENO, 2003).

O triple-DES ou também conhecido como 3DES faz o processo de cifragem idêntico ao algoritmo DES, só que é repetido três vezes, podendo usar sempre o mesmo valor da chave ou usando duas ou três chaves diferentes.

Em 15 de Maio de 1973, durante o governo de Richard Nixon, o NBS publicou uma notícia solicitando formalmente propostas de algoritmos criptográficos para proteger dados durante transmissões e armazenamento. A notícia explicava porque a cifragem de dados era um assunto importante.

O NBS solicitava técnicas e algoritmos para a cifragem de dados por computador. Também solicitava técnicas para implementar as funções criptográficas de gerar, avaliar e proteger chaves criptográficas; manter arquivos cifrados com chaves que expiram; fazer atualizações parciais em arquivos cifrados e misturar dados cifrados com claros para permitir marcações, contagens, roteamentos, etc. O NBS, no seu papel de estabelecer padrões e de auxiliar o governo e a indústria no acesso à tecnologia, se encarregaria de avaliar os métodos de proteção para preparar as linhas de ação.

O NBS ficou esperando por respostas. Nenhuma apareceu até 6 de Agosto de 1974, três dias antes da renúncia de Nixon, quando a IBM apresentou um algoritmo candidato que ela havia desenvolvido internamente, denominado Lúcifer.

Após avaliar o algoritmo com a ajuda da *National Security Agency* (NSA), o NBS adotou o algoritmo Lúcifer com algumas modificações sob a denominação de *Data Encryption Standard* - DES em 15 de Julho de 1977 (TKOTZ, 2003).

O DES foi rapidamente adotado na mídia não digital, como nas linhas telefônicas públicas. Depois de alguns anos, a *International Flavors and Fragrances* estava utilizando o

DES para proteger as transmissões por telefone das suas preciosas fórmulas de sabores e fragrâncias.

Neste meio tempo, a indústria bancária, a maior usuária de criptografia depois do governo, adotou o DES como padrão para o mercado bancário atacadista. Os padrões do mercado atacadista da indústria bancária são estabelecidos pelo *American National Standards Institute* (ANSI). A norma ANSI X3.92, adotada em 1980, especificava o uso do algoritmo DES (TKOTZ, 2003).

O algoritmo DES é composto por operações simples como: permutações, substituições, XOR e deslocamentos. O DES criptografa informações através do processo de cifra de bloco com tamanho de 64 bits e retorna blocos de texto cifrado do mesmo tamanho usando uma chave de 56 bits.

O processo principal do algoritmo é executado 16 vezes, em cada iteração é utilizada uma sub-chave derivada da chave original.

2.4.3 – Algoritmo Simétrico AES

Com a evolução das máquinas uma chave de 56 bits, como a usada pelo algoritmo DES, não poderia mais garantir a segurança desejada por muito tempo. Em 1997 iniciou-se o processo de escolha do sucessor do DES, o *Advanced Encryption Standard* - AES. O NIST (*National Institute of Standards and Technology*) especificou que os candidatos deveriam operar com tamanho de chave e bloco variável com tamanho mínimo de 128 bits.

O NIST colocou como requisitos fundamentais:

- **Segurança Forte**: O algoritmo projetado devia suportar os ataques futuros;

- **Projeto Simples**: Facilitando a análise e certificação matemática da segurança oferecida pelo algoritmo;
- **Desempenho**: Razoavelmente bom em uma variedade de plataformas, variando de smartcards a servidores;
- **Não serem patenteados**: Os algoritmos deviam ser de domínio público e deviam estar disponíveis mundialmente.

No primeiro congresso, chamado também de rodada, o NIST recebeu vinte e uma submissões, das quais quinze atendiam às exigências. Estes algoritmos foram testados e avaliados pela comunidade científica e empresas ligadas ao ramo de segurança durante o intervalo de tempo até a segunda rodada.

Para o segundo congresso foram escolhidos apenas cinco algoritmos (AESWINNER, 2003): MARS, RC6, Rijndael, Serpent e TwoFish. Esses algoritmos satisfaziam as principais condições, entre outras, de: cifrar e decifrar blocos de 128 bits; trabalhar com chaves de 128/192/256 bits; ser mais rápido que o 3DES.

Como todos os requisitos básicos foram supridos pelos concorrentes, a decisão foi tomada com base em certas características, tais como: segurança, eficiência em hardware e software, flexibilidade de implementação e modos de operação.

O algoritmo vencedor foi o Rijndael (AESWINNER, 2003), sendo a decisão informada oficialmente no dia 2 de outubro de 2000. Desse momento em diante o cifrador Rijndael passou a ser chamado de AES. A escolha do cifrador Rijndael contra os outros quatro finalistas do processo, os quais também foram classificados como altamente seguros pelo NIST, foi baseada na eficiência e baixa requisição de recursos. Os demais cifradores finalistas do concurso para AES podem ser utilizados em produtos específicos (MIERS, 2002).

O AES é um cifrador de bloco com tamanho de bloco e chave variáveis entre 128, 192 e 256 bits. Isto significa que se pode ter tamanho de blocos com tamanhos de chaves diferentes. Em função do tamanho de bloco e chaves é determinada a quantidade de rodadas necessárias para cifrar/decifrar.

O AES opera desta forma com um determinado número de blocos de 32 bits, os quais são ordenados em colunas de 4 bytes, as quais são chamadas de Nb. Os valores de Nb possíveis são de 4, 6 e 8 equivalentes a blocos de 128, 192 e 256 bits.

Assim sempre que for referido Nb, significa que tem-se Nb x 32 bits de tamanho de bloco de dados. A chave é agrupada da mesma forma que o bloco de dados, isto é, em colunas, sendo representado pela sigla Nk. Com base nos valores que Nb e Nk podem assumir é que se determina a quantidade de rodadas a serem executadas, identificada pela sigla Nr. Através dos dados contidos na Tabela 8 pode-se verificar as possíveis combinações e o número de rodadas necessárias na execução do algoritmo AES. O processo de cifrar e decifrar no AES não são funções idênticas, como ocorre na maioria dos cifradores.

Tabela 8 – Nr em função do tamanho de bloco e chave no AES

Nr	Nb=4	Nb=6	Nb=8
Nk=4	10	12	14
Nk=6	12	12	14
Nk=8	14	14	14

2.4.4 – Algoritmo Assimétrico RSA

O RSA é um sistema de criptografia de chave assimétrica ou criptografia de chave pública que foi inventado por volta de 1977 pelos professores do MIT (*Massachsetts Institute*

of Technology) Ronald Rivest, Adi Shamir e o professor da USC (*University of Southern California*) Leonard Adleman (RIVEST, et. al., 1978).

O sistema consiste em gerar uma chave pública (geralmente utilizada para cifrar os dados) e uma chave privada (utilizada para decifrar os dados) através de números primos grandes, o que dificulta a obtenção de uma chave a partir da outra.

Quanto maiores forem os números primos utilizados para a criação da chave, maior é a segurança proporcionada por esse algoritmo. Hoje em dia os números primos que são utilizados têm geralmente 512 bits de comprimento e combinados formam chaves de 1024 bits. Em algumas aplicações como, por exemplo, bancárias que exigem o máximo de segurança a chave chega a ser de 2048 bits (MORENO, et. al., 2005).

Com o passar do tempo, a tendência é que o comprimento da chave aumente cada vez mais. Esse fenômeno acontece, em grande parte, pelo avanço nos sistemas computacionais que acompanham o surgimento de computadores que são capazes de fatorar chaves cada vez maiores em um tempo cada vez menor.

Os algoritmos para a geração da chave pública e privada usadas para cifrar e decifrar as mensagens são simples. Observe-os a seguir:

1) Escolhe-se dois números primos grandes (**p** e **q**);

2) Gera-se um número **n** através da multiplicação dos números escolhidos anteriormente (**n = p . q**);

3) Escolhe-se um número **d**, tal que **d** é menor que **n** e **d** é relativamente primo à **(p-1).(q-1)**;

4) Escolhe-se um número **e** tal que (**ed-1**) seja divisível por **(p-1).(q-1)**. Para realizar esse cálculo é necessário o algoritmo de Euclides (TERADA, 2000).

Os valores **e** e **d** são chamados de expoentes público e privado, respectivamente. O par **(n,e)** é a chave pública e o par **(n,d)** é a chave privada. Os valores **p** e **q** devem ser mantidos em segredo ou destruídos.

Para cifrar uma mensagem com esse algoritmo é realizado o seguinte cálculo:

$C = T^e \bmod n$, onde **C** é a mensagem cifrada, **T** é o texto original, **e** e **n** são dados a partir da chave pública **(n,e)**.

A única chave que pode decifrar a mensagem **C** é a chave privada **(n,d)** através do calculo de: **$T = C^d \bmod n$**.

Uma implementação do algoritmo RSA foi realizada em linguagem C e em hardware. O gráfico da Figura 3 mostra que o algoritmo RSA é o mais lento de todos os algoritmos apresentados. Isto ocorre pois o RSA é um algoritmo assimétrico (possui duas chaves, uma para a cifragem e outra para a decifragem).

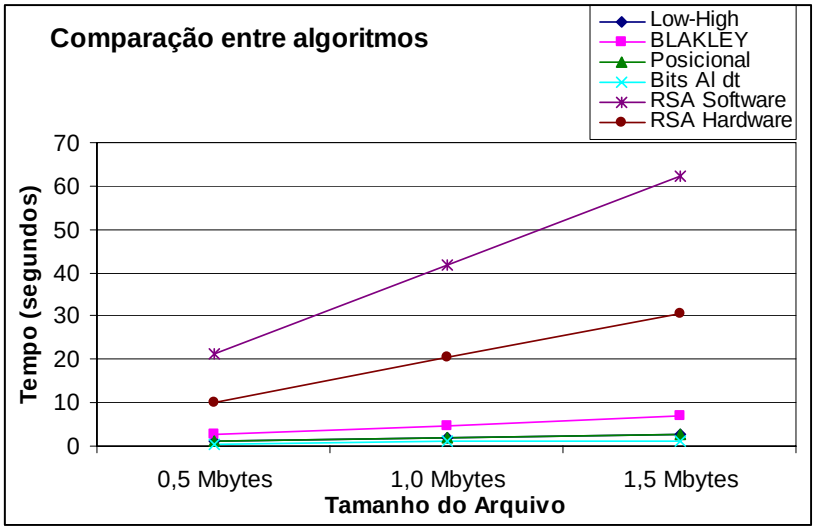

Figura 3 – Comparação dos algoritmos Simétricos x Assimétricos (CHIARAMONTE, 2003)

Também pode-se notar que o algoritmo RSA implementado em Software é mais lento que o algoritmo RSA implementado em Hardware, no entanto, a implementação em

hardware do algoritmo RSA ainda é mais lenta que os demais algoritmos simétricos apresentados. No gráfico: Low-High e Blakley se referem às implementações do IDEA; Posicional e Posicional Al. Dt. são implementações do algoritmo Posicional, a primeira é uma versão simples do algoritmo Posicional e a segunda uma versão com o acréscimo de bits aleatórios distribuídos que podem ser encontrados em (CHIARAMONTE, 2003).

Os desempenhos foram obtidos com base em um computador Duron 950Mhz e 128 Mbytes de memória utilizando o sistema operacional Windows 98.

2.5 – Funções de *Hash*

Funções de *hash* ou funções de espalhamento são funções que calculam um valor **y** de comprimento relativamente menor que o comprimento de um texto **x**. Esse valor **y** é chamado de MAC (*Message Authentication Code*) ou *Message Digest* de **x**.

Essas funções são utilizadas para garantir a integridade da mensagem, já que o objetivo dessas funções é gerar um valor **y** diferente para cada mensagem; com isso, caso a mensagem **x** for modificada para uma mensagem **x'** quando o destinatário receber a mensagem **x'** ele irá recalcular **y** para a mensagem que recebeu (**x'**). Como cada mensagem gera um **y** diferente ele irá detectar que **y** para a mensagem que ele recebeu é diferente do que ele esperava e com isso detectar que a mensagem foi alterada.

No entanto, existe uma probabilidade de que ao calcular a função para mensagens diferentes o valor **y** obtido seja o mesmo. Essa situação é chamada de colisão (MENEZES, 1997).

Portanto o objetivo dos projetistas das funções de espalhamento é minimizar a probabilidade de ocorrer colisões.

São exemplos de funções de espalhamento os algoritmos MD4 (*Message Digest 4*), MD5 e SHS (*Secure Hash Standard*). O SHS é um padrão que consiste de vários algoritmos, entre eles está presente o algoritmo SHA-1 que é apresentado neste capítulo.

2.5.1 – Algoritmo MD5

O algoritmo MD5 foi proposto por Rivest e descrito no RFC1321 em Abril de 1992 (MORENO, et. al., 2005). Ele foi desenvolvido para executar rapidamente em máquinas de 32 bits e não requer longas tabelas de substituição, o que permite que seja codificado de forma compacta (RFC1321, 1992).

O MD5 é uma extensão do algoritmo MD4, sendo cerca de 30% mais lento (TERADA, 2000), mas garantindo uma maior segurança. O algoritmo MD5 possui probabilidade de colisão a cada 2^{64} aplicações contra 2^{20} aplicações do algoritmo MD4 (MENEZES, 1997).

No algoritmo MD5 a entrada pode ser de comprimento arbitrário e o processamento ocorre em blocos de 512 bits e a saída é de 128 bits.

Inicialmente, deve-se realizar o preenchimento da mensagem, para que ela fique de tamanho congruente a 448 (mod 512), ou seja, ao separar a mensagem em blocos de 512 bits no último bloco restará exatamente 448 bits. O formato do preenchimento a ser acrescentado é um bit ‘1’ seguido de bits ‘0’ até que o tamanho seja congruente a 448 (mod 512). Importante ressaltar que mesmo que a mensagem original seja de tamanho 448 (mod 512) é necessário

realizar o preenchimento, com isto, no mínimo 1 e no máximo 512 bits serão acrescentados à mensagem.

Como descrito anteriormente, o algoritmo MD5 processa blocos de 512 bits, com isto é necessário que o tamanho da mensagem seja múltiplo de 512. Como após o preenchimento o tamanho do ultimo bloco é de 448 bits são necessários 64 bits para completar o bloco. Esses últimos 64 bits são preenchidos com o tamanho em bits da mensagem original. Caso o tamanho da mensagem não possa ser representado em 64 bits, apenas os bits menos significativos são considerados.

Após realizar as operações descritas acima, a mensagem está no formato correto para ser processada. Porém, antes de iniciar o processamento é necessário inicializar os buffers do MD5. O algoritmo MD5 utiliza quatro buffers de 32 bits cada, aqui chamados de A, B, C e D; para realizar o processamento e o cálculo do *hash*. Os buffers do algoritmo MD5 devem ser inicializados com os valores descritos na Tabela 9 (os valores estão em hexadecimal).

Importante salientar que o algoritmo MD5 foi criado para uma máquina *little-endian*, desta forma deve-se considerar primeiro os bytes menos significativos (MORENO, et. al., 2005).

Tabela 9 – Valores para preencher os Buffers do MD5 (RFC1321, 1992)

A	01	23	45	67
B	89	AB	CD	EF
C	FE	DC	BA	98
D	76	54	32	10

Para realizar o processamento, o bloco de entrada de 512 bits é dividido em 16 partes de 32 bits aqui chamadas de x_0, x_1,..., x_{15}. Observe que 16 x 32 = 512, que é o tamanho do bloco usado no MD5.

Inicialmente é realizada a cópia dos valores antigos do buffer e então são realizadas operações sobre o texto de entrada alterando os valores armazenados no buffer (que está composto por quatro partes A, B, C e D).

As operações realizadas são divididas em quatro passos. Cada passo utiliza uma função auxiliar diferente e utiliza uma ordem diferente para aplicar as entradas x_0, x_1,..., x_{15}. A Tabela 10 sumariza quais são as funções auxiliares utilizadas em cada passo bem como a ordem de aplicação das partes do texto de entrada (RFC1321, 1992).

Tabela 10 – Sumário das operações realizadas no MD5 (RFC1321, 1992).

Passo	Função Auxiliar/Operação
1	F(X,Y,Z) = XY or not(X) Z **[abcd k s i] => a = b + ((a + F(b,c,d) + X[k] + T[i]) <<< s)** [ABCD 0 7 1] [DABC 1 12 2] [CDAB 2 17 3] [BCDA 3 22 4] [ABCD 4 7 5] [DABC 5 12 6] [CDAB 6 17 7] [BCDA 7 22 8] [ABCD 8 7 9] [DABC 9 12 10] [CDAB 10 17 11] [BCDA 11 22 12] [ABCD 12 7 13] [DABC 13 12 14] [CDAB 14 17 15] [BCDA 15 22 16]
2	G(X,Y,Z) = XZ or Y not(Z) **[abcd k s i] => a = b + ((a + G(b,c,d) + X[k] + T[i]) <<< s)** [ABCD 1 5 17] [DABC 6 9 18] [CDAB 11 14 19] [BCDA 0 20 20] [ABCD 5 5 21] [DABC 10 9 22] [CDAB 15 14 23] [BCDA 4 20 24] [ABCD 9 5 25] [DABC 14 9 26] [CDAB 3 14 27] [BCDA 8 20 28] [ABCD 13 5 29] [DABC 2 9 30] [CDAB 7 14 31] [BCDA 12 20 32]
3	H(X,Y,Z) = X xor Y xor Z **[abcd k s t] => a = b + ((a + H(b,c,d) + X[k] + T[i]) <<< s)** [ABCD 5 4 33] [DABC 8 11 34] [CDAB 11 16 35] [BCDA 14 23 36] [ABCD 1 4 37] [DABC 4 11 38] [CDAB 7 16 39] [BCDA 10 23 40] [ABCD 13 4 41] [DABC 0 11 42] [CDAB 3 16 43] [BCDA 6 23 44] [ABCD 9 4 45] [DABC 12 11 46] [CDAB 15 16 47] [BCDA 2 23 48]
4	I(X,Y,Z) = Y xor (X or not(Z)) **[abcd k s t] => a = b + ((a + I(b,c,d) + X[k] + T[i]) <<< s)** [ABCD 0 6 49] [DABC 7 10 50] [CDAB 14 15 51] [BCDA 5 21 52] [ABCD 12 6 53] [DABC 3 10 54] [CDAB 10 15 55] [BCDA 1 21 56] [ABCD 8 6 57] [DABC 15 10 58] [CDAB 6 15 59] [BCDA 13 21 60] [ABCD 4 6 61] [DABC 11 10 62] [CDAB 2 15 63] [BCDA 9 21 64]

Importante ressaltar que o algoritmo MD5 utiliza 64 constantes, aqui chamadas de T[1], T[2],...,T[64]. Essas constantes são obtidas através do calculo: 2^{32} * abs(sen(i)). Para evitar que a cada bloco seja necessário realizar tal operação é recomendável, em uma

implementação, criar uma tabela em que os valores calculados estejam previamente armazenados e simplesmente utilizá-los. A Tabela 11 apresenta os valores destas constantes.

Após o calculo de todas as operações de um determinado bloco é necessário realizar o somatório dos valores antigos contidos no buffer com os valores atuais antes de realizar o processo no próximo bloco. Após realizar o processamento de todos os blocos, o valor de saída é o conteúdo do buffer do MD5 (A, B, C e D).

Tabela 11 – Constantes utilizadas pelo MD5 (RFC1321, 1992)

	1	2	3	4
1	0xd76aa478	0xe8c7b756	0x242070db	0xc1bdceee
5	0xf57c0faf	0x4787c62a	0xa8304613	0xfd469501
9	0x698098d8	0x8b44f7af	0xffff5bb1	0x895cd7be
13	0x6b901122	0xfd987193	0xa679438e	0x49b40821
17	0xf61e2562	0xc040b340	0x265e5a51	0xe9b6c7aa
21	0xd62f105d	0x02441453	0xd8a1e681	0xe7d3fbc8
25	0x21e1cde6	0xc33707d6	0xf4d50d87	0x455a14ed
29	0xa9e3e905	0xfcefa3f8	0x676f02d9	0x8d2a4c8a
33	0xfffa3942	0x8771f681	0x6d9d6122	0xfde5380c
37	0xa4beea44	0x4bdecfa9	0xf6bb4b60	0xbebfbc70
41	0x289b7ec6	0xeaa127fa	0xd4ef3085	0x04881d05
45	0xd9d4d039	0xe6db99e5	0x1fa27cf8	0xc4ac5665
49	0xf4292244	0x432aff97	0xab9423a7	0xfc93a039
53	0x655b59c3	0x8f0ccc92	0xffeff47d	0x85845dd1
57	0x6fa87e4f	0xfe2ce6e0	0xa3014314	0x4e0811a1
61	0xf7537e82	0xbd3af235	0x2ad7d2bb	0xeb86d391

O algoritmo MD5 foi implementado em linguagem C e alguns dados de desempenho são apresentados na seção 2.6.

2.5.2 – Algoritmo SHA-1

O algoritmo SHA-1 foi desenvolvido pelo NIST em 1993 (RFC3174, 1994). A entrada pode ser de comprimento arbitrário e deve ser completada para que o comprimento se torne múltiplo de 512 bits, como ocorre com o MD5 (MORENO, et. al., 2005).

A saída deste algoritmo é de 160 bits sendo que blocos de 512 bits são processados a cada passo (RFC3174, 1994). Este algoritmo possui uma menor probabilidade de colisão quando comparado com o algoritmo MD5, sendo que no algoritmo SHA-1 existe a probabilidade de ocorrer colisões após 2^{80} aplicações, contra 2^{64} aplicações do MD5 (MENEZES, 1997).

A estrutura do algoritmo SHA-1 é parecida com a estrutura do algoritmo MD5 (MORENO, et. al., 2005). O algoritmo SHA-1, a exemplo do algoritmo MD5, possui um buffer que é atualizado a cada operação e que será o resultado final do *hash*. Neste caso como a saída é de 160 bits, o buffer é composto por cinco partes de 32 bits aqui chamados de (A, B, C, D e E). O algoritmo SHA-1 utiliza também um segundo buffer de cinco partes de 32 bits que aqui são chamadas de H_0, H_1, H_2, H_3 e H_4.

O buffer do algoritmo SHA-1 deve ser iniciado com os valores contidos na Tabela 12 (os valores estão em hexadecimal).

Tabela 12 – Valores para preencher o buffer do SHA-1 (RFC3174, 1994)

H0	67	45	23	01
H1	EF	CD	AB	89
H2	98	BA	DC	FE
H3	10	32	54	76
H4	C3	D2	E1	F0

No inicio de cada passo, um bloco de entrada (X) de 512 bits é utilizado para criar um vetor W de 80 palavras de 32 bits. Inicialmente, o bloco de entrada X é dividido para formar as 16 primeiras palavras do vetor W (W_0 a W_{15}). As palavras de W_{16} até W_{79} são formadas obedecendo: $W_i = (W_{i-3}$ XOR W_{i-8} XOR W_{i-14} XOR $W_{i-16})$ <<< 1; onde **a** <<< **b** representa o deslocamento circular de **a** para esquerda **b** bits.

O algoritmo SHA-1 possui quatro funções e quatro constantes que são utilizadas de acordo com a iteração que estiver sendo processada; desta forma a Tabela 13 apresenta as funções e constantes utilizadas e indica em quais iterações devem ser utilizadas.

Tabela 13 – Funções e Constantes utilizadas pelo SHA-1 (RFC3174, 1994)

Função	Constante	Iterações
f(B,C,D) = (B AND C) OR ((NOT B) AND D)	5A827999	0 a 19
f(B,C,D) = B XOR C XOR D	6ED9EBA1	20 a 39
f(B,C,D) = (B AND C) OR (B AND D) OR (C AND D)	8F1BBCDC	40 a 59
f(B,C,D) = B XOR C XOR D	CA62C1D6	60 a 79

Antes de executar as iterações necessárias para o processamento de um bloco com o algoritmo SHA-1, é realizada uma cópia dos valores contidos no buffer H_0, H_1, H_2, H_3 e H_4 para o buffer A, B, C, D e E.

O processamento de um bloco com o algoritmo SHA-1 consiste de 80 iterações (de i=0 até i=79) em que são processadas as seguintes operações:

```
TEMP = (A<<<5) + fi(B,C,D) + E + Wi + Ki;
E = D; D = C; C = (B<<<30); B = A; A = TEMP;
```

Após a execução do laço principal é realizado o somatório dos dois buffers como segue:

```
H0 = H0 + A; H1 = H1 + B; H2 = H2 + C; H3 = H3 + D; H4 = H4 + E
```

Após processar todos os blocos da mensagem, o valor do hash estará armazenado no buffer H_0, H_1, H_2, H_3 e H_4.

Assim como o algoritmo MD5, o algoritmo SHA-1 também foi implementado em linguagem C e seus dados de desempenho são apresentados na seção 2.6.

2.6 – Dados sobre Desempenho dos Algoritmos Implementados

Foram implementados alguns outros algoritmos em linguagem C, são eles: os algoritmos simétricos DES (FIPS-PUB46-3, 1999) e AES (DAEMEN e RIJMEN, 1999); e os algoritmos de hash MD5 e SHA-1.

Para alguns testes mais detalhados, um arquivo de vídeo de 5 (cinco) minutos de duração foi compactado utilizando taxa de bits variável (VBR – *Variable Bit Rate*) com quatro taxas médias de compactação diferentes. Para a compactação foi utilizado o *codec* de vídeo MPEG2 (ROSE, 1995). A Tabela 14 apresenta um resumo onde consta a taxa de compactação média escolhida e o tamanho final do arquivo.

Tabela 14 – Taxa média de compactação e tamanhos finais dos arquivos

Taxa de Compactação	Tamanho (Bytes)
64 kbps	16.522.240
450 kbps	21.791.744
760 kbps	33.224.704
1,5 mbps	60.607.488

A Tabela 15 mostra o tempo gasto na criptografia dos arquivos. Os tempos apresentados estão em segundos. Através desta tabela pode-se perceber que o algoritmo RSA necessitou um tempo maior que o tempo do vídeo (5 minutos = 300 segundos) para compactar um arquivo utilizando taxa de compactação média de 1,5 mbps. Isto significa que não seria possível utilizar o algoritmo RSA desta forma para aplicações em tempo real. Importante enfatizar que somente o tempo de criptografia foi considerado. Desta forma, seria interessante que no futuro se realizassem testes em tempo real considerando também o tempo de compactação.

Tabela 15 – Tempo gasto na criptografia (segundos)

	Tempo de Criptografia em Segundos				
Vídeo	AES	DES	RSA	MD5	SHA-1
64 kbps	6,469	44,641	133,266	0,906	0,937
450 kbps	8,609	58,859	174,719	1,578	1,218
760 kbps	13,656	89,734	266,828	2,203	1,853
1,5 mbps	25,109	163,671	488,016	3,563	3,484

A Figura 4 apresenta os dados comparativos relativos aos algoritmos simétricos, comparando com o algoritmo assimétrico RSA. Os algoritmos simétricos utilizados foram o DES com chave de 64 bits e AES com chave de 128 bits.

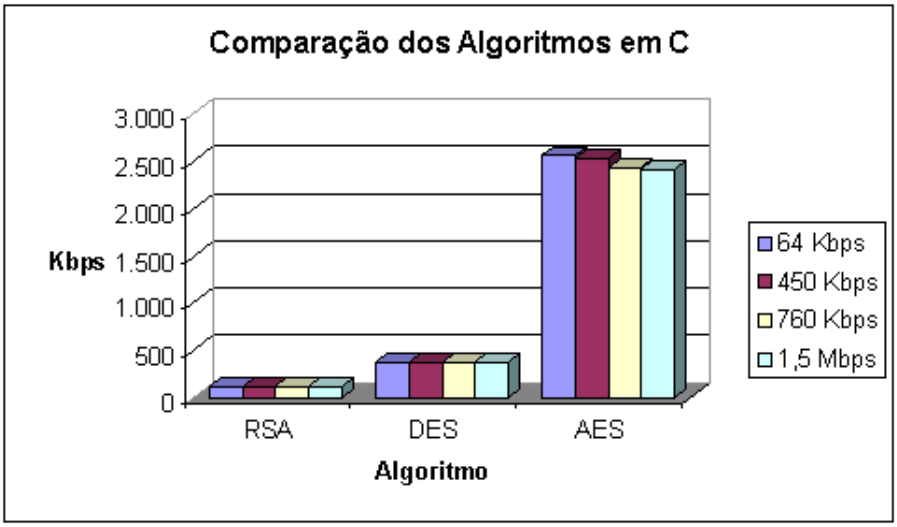

Figura 4 – Comparação dos Algoritmos em C

A Figura 5 apresenta os dados sobre as implementações dos algoritmos de *hash*. Importante enfatizar que os algoritmos de *hash* são mais rápidos que os algoritmos criptográficos realizando a operação a uma velocidade média acima de 10.000 Kbps.

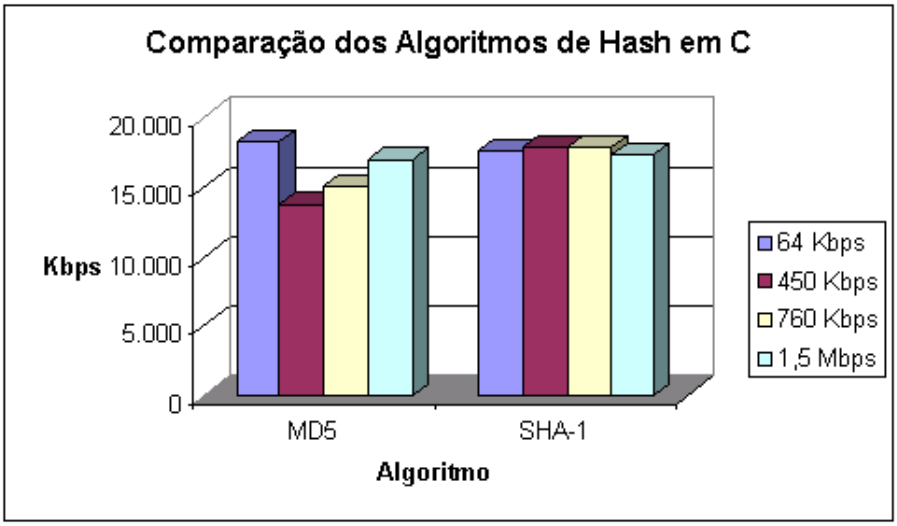

Figura 5 – Comparação dos algoritmos de Hash em C

2.7. Desempenho de outras implementações

A partir dos testes realizados e apresentados na seção anterior, foi também realizado um levantamento sobre o desempenho de outras implementações disponíveis. Brian Gladman (GLADMAN, 2006) disponibilizou implementações em linguagem C dos algoritmos participantes do processo de escolha do AES, que são: Rijndael (vencedor do concurso e classificado como novo padrão – AES), RC6, Serpent, Twofish, MARS.

Com base nestas implementações foram realizados testes de desempenho para verificar as diferenças de desempenho entre algoritmos e também entre os resultados obtidos na seção anterior. Os dados de desempenho destas implementações são apresentados na Figura 6.

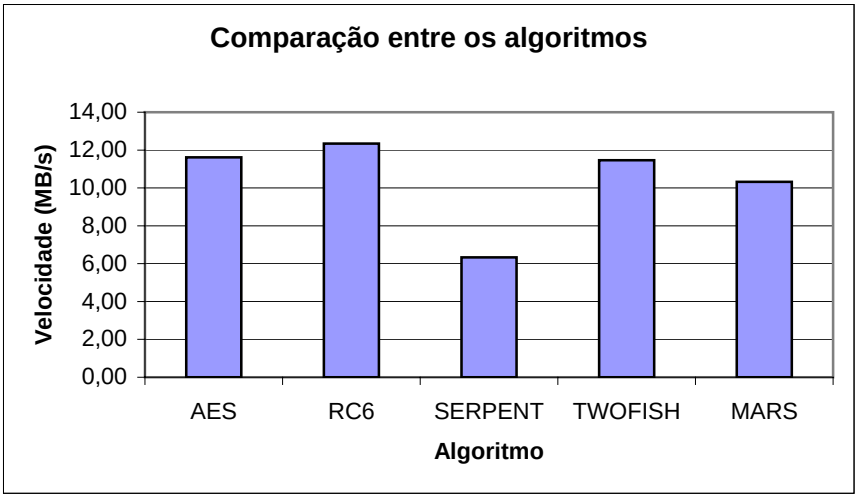

Figura 6 – Comparação de desempenho entre os algoritmos

Outra opção de implementação disponível, que foi utilizada no projeto, foram as implementações desenvolvidas em linguagem Java do AES (Rijndael) e DES presentes na arquitetura JCA. Os dados de desempenho destas implementações são apresentados na Figura 7. Pode-se perceber observando os gráficos da Figura 6 e da Figura 7 que o algoritmo AES implementado utilizando a JCA teve um menor desempenho quando comparado com a implementação realizada por Brian Gladman.

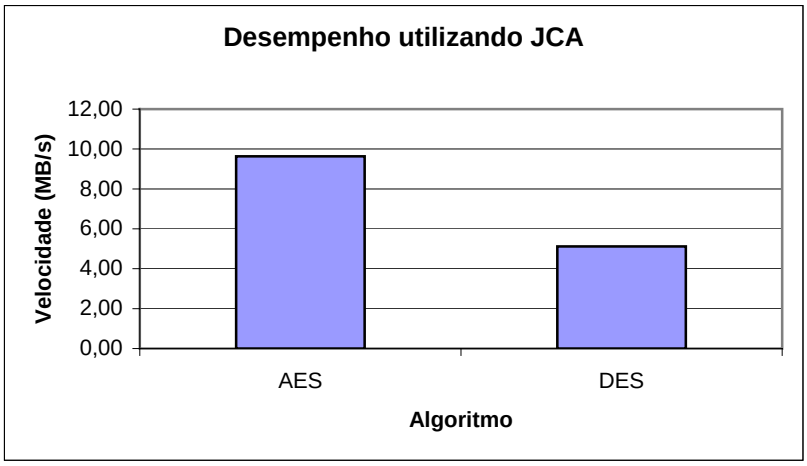

Figura 7 – Desempenho utilizando JCA

Desta forma, testes foram realizados utilizando essas implementações e verificou-se que seu desempenho é muito superior que as apresentadas na seção anterior. Importante destacar que todos os testes foram realizados com um computador Celeron 2.4 Ghz com 256Mb de memória RAM utilizando o sistema operacional Microsoft Windows XP.

Sendo assim, optou-se por utilizar as implementações presentes na JCA e as de Brian Gladman no desenvolvimento do sistema.

Como as implementações do Brian Gladman foram disponibilizadas em linguagem C e o sistema foi implementado em Java, foi necessário utilizar a tecnologia de JNI (*Java Native Interface*) para realizar a interligação entre as duas linguagens.

CAPÍTULO 3 – REDES, QoS E SEGURANÇA

Este capítulo apresenta os principais conceitos relacionados com redes de computadores e Qualidade de Serviço relacionando-os com aspectos sobre segurança em redes.

3.1 – Breve Histórico Sobre Redes

As redes de comunicação permitem a troca de informações em forma de voz, vídeo, e-mail ou arquivos de computador (WALRAND e VARAIYA, 2000).

As primeiras redes de comunicação foram as redes de telefonia que surgiram por volta de 1890 e operavam através de chaveamento manual (WALRAND e VARAIYA, 2000). A partir daí a evolução das redes de comunicação foi constante.

No final da década de 1960, o DARPA (*U.S. Department of Defense Advanced Research Projects Agency*) promoveu o desenvolvimento de uma rede de pacotes chaveados, que veio a ser chamada de ARPANET. A ARPANET iniciou suas operações em 1969 conectando quatro computadores. Surgem aí as redes de computadores.

Segundo Tanenbaum (1996), uma rede de computadores é uma coleção de computadores autônomos interconectados. Hoje em dia as redes de computadores são populares e funcionam sobre diversos tipos de tecnologias. Neste contexto, é importante destacar a Internet, que é um conjunto de redes de computadores, cada qual com suas tecnologias, todas interconectadas através de *gateways*, que são os responsáveis pelas

conversões entre essas diferentes tecnologias. Algumas das principais tecnologias utilizadas em redes de computadores são apresentadas na seção 3.2.

Normalmente, as informações que trafegam na internet são entregues o mais rápido possível, seguindo a política do melhor esforço. Entretanto, não há garantias de que uma determinada informação será entregue obedecendo a um determinado intervalo de tempo, ou mesmo garantias de que ela será entregue. Em algumas situações em que há congestionamento, alguns pacotes são descartados sem distinção, não existindo garantia de que o serviço será concluído com sucesso (SANTOS, 1999). Existem alguns serviços, como aplicações de videoconferência, em que a garantia de entrega das informações no tempo adequado é importante; em outros serviços, como por exemplo, o correio eletrônico, o tempo para a entrega de tal informação não é tão crítico. Esta diferença entre o comportamento e as necessidades das aplicações foi o que motivou a criação de um conceito chamado de QoS (XIAO e NI, 1999) que é apresentado em mais detalhes na seção 3.3.

3.2 – Tecnologias e Protocolos de Rede

Esta seção apresenta alguns conceitos relacionados ao hardware e software de rede descrevendo os conceitos de hierarquia de protocolos. Também são descritos alguns aspectos referentes à camada física de rede, relacionando-a com a segurança.

3.2.1 – Conceitos Básicos

Segundo Tanenbaum (1996) não existe uma taxonomia em que se encaixem todos os tipos de rede, mas existem duas dimensões importantes: tecnologia de transmissão e escala.

Existem basicamente dois tipos de tecnologias de transmissão (TANENBAUM, 1996): redes *broadcast* e redes ponto a ponto.

Nas redes *broadcast* existe um único canal que é compartilhado por todas as máquinas da rede. As mensagens enviadas para qualquer uma das máquinas são recebidas por todas as outras. Desta forma, quando uma máquina recebe uma mensagem, esta deve verificar o campo de endereço contido no pacote para verificar se a mensagem é ou não para ela. Caso seja ela deverá realizar o tratamento necessário, mas caso não seja, deverá descartar a mensagem.

Os sistemas de *broadcast* permitem endereçar, através de endereços especiais, um pacote para todas as máquinas da rede, ou para um subconjunto de máquinas da rede. Estas práticas são conhecidas como *broadcasting* e *multicasting* respectivamente.

As redes ponto a ponto consistem na existência de diversas conexões entre pares de máquinas. Desta forma, um pacote neste tipo de rede deve passar por uma ou mais máquinas intermediárias antes de chegar ao destino. Isto permite que entre duas máquinas existam rotas diferentes, com distâncias diferentes; o que torna muito importante o uso de algoritmos de roteamento.

Quanto à escala, as redes podem ser divididas em: redes locais (LAN – *Local Area Networks*), redes metropolitanas (MAN – *Metropolitan Area Networks*) e redes amplas ou de larga escala (WAN – *Wide Area Networks*) (TANENBAUM, 1996).

3.2.2 – Alguns Aspectos sobre Software de Rede

O software de rede é organizado e estruturado através de camadas de protocolos. Tal estrutura é chamada de Hierarquia de Protocolos. Esta divisão em camadas provê modularização e reduz a complexidade de desenvolvimento.

Segundo Tanenbaum (1996), o número de camadas, o nome, o conteúdo e a função de cada camada difere de um tipo de rede para outro. No entanto, em todos os tipos de rede, o propósito de cada camada é fornecer serviços para as camadas mais altas, poupando-as dos detalhes de como os serviços das camadas mais baixas foram implementados.

A camada *n* de uma máquina se comunica com a camada *n* de outra máquina utilizando um conjunto de regras e convenções conhecidas como protocolo (TANENBAUM, 1996).

3.2.3 – Meios de transmissão

Geralmente a camada mais baixa na hierarquia de protocolos é a camada física. Esta camada é responsável em definir como os dados são transmitidos fisicamente, na forma de seqüência de bits (TANENBAUM, 1996).

Para realizar esta transmissão é necessário um meio de transmissão. Existem vários meios para a transmissão de dados. Aqui são apresentados alguns principais meios de transmissão relacionando-os com a segurança oferecida.

Um dos meios mais utilizado é o cabo de par trançado. Este meio normalmente é utilizado em redes do tipo *Ethernet* em que os elementos são interligados através de *hubs* ou *switches* que servem como elementos centralizadores. Um aspecto importante para a segurança destas redes está relacionado com o uso destes equipamentos. Quando um *hub* é instalado em uma rede *Ethernet,* os dados são transmitidos fisicamente em *broadcast*; desta forma, uma máquina configurada para trabalhar em modo promíscuo pode capturar todos os pacotes que passam por esta rede. No entanto, utilizar *switches* como elementos centralizadores é uma boa estratégia para se conseguir uma melhora de performance e segurança. Um *switch* é capaz de avaliar o endereço físico das máquinas envolvidas na rede e uma vez que um dado seja enviado para uma determinada máquina, o *switch* se encarregará de encaminhá-lo para o endereço físico desta máquina. Desta forma, as outras máquinas na rede não recebem a informação. Isto, além de evitar o tráfego de pacotes desnecessários na rede, aumentando o desempenho, evita que uma máquina configurada para trabalhar em modo promíscuo capture todos os pacotes que passam pela rede. No entanto, ainda assim é difícil controlar capturas de dados indesejáveis na rede.

A tecnologia de transmissão por fibra óptica é um outro exemplo de meio de transmissão. Esta tecnologia, além de muito rápida, é uma das que oferece maior segurança, uma vez que, é muito difícil capturar dados de uma rede deste tipo sem autorização (TANENBAUM, 1996).

Outra tecnologia de transmissão que tem obtido muita importância ultimamente é a tecnologia de redes sem fio, ou redes *wireless*. Neste tipo de tecnologia existem protocolos especiais para a segurança, como por exemplo, o protocolo WEP (*Wireless Encryption Protocol*). No entanto, a segurança oferecida pelas redes *wireless* ainda possui algumas fraquezas (BORISOV et. al., 2001), o que exige mais segurança adicional quando este tipo de tecnologia é utilizado.

Existem ainda outras tecnologias, como o PLC (*Power Line Communication*), em que sinais de dados são transmitidos através das linhas de eletricidade. Neste sistema há uma grande necessidade de segurança uma vez que os sinais podem ser capturados por qualquer equipamento ligado na rede elétrica dentro da área de cobertura da rede PLC. Alguns equipamentos para este tipo de rede, como o PowerPacket da Intellon, possuem softwares para criptografia incluídos, entretanto, a manipulação das chaves não é tão flexível e é restrita a sistemas que utilizam o sistema operacional Windows (INTELLON, 2005).

3.3 – QoS (*Quality of Service*)

Existem atualmente diversos serviços funcionando sobre a Internet. Dentre eles, podem ser destacados: serviços de páginas, serviços de correio eletrônico (e-mail), serviços de vídeo conferência, serviços de voz sobre IP (VoIP), etc.

Cada tipo de serviço possui características diferentes quanto à utilização da rede. Tais características estão relacionadas com: o tráfego gerado pela aplicação, tempo de atraso aceitável, taxa de perda de pacotes aceitável, etc.

As informações que as aplicações geram podem assumir diversos formatos como: texto, voz, áudio, dados, gráficos, figuras, animações e vídeo. Outra característica das informações geradas pelas aplicações é o sentido em que as informações trafegam: em um único sentido; em duplo sentido; *broadcast*; e multiponto (WALRAND e VARAIYA, 2000).

O tráfego gerado pelas aplicações pode ser classificado em três tipos (WALRAND e VARAIYA, 2000):

- *Constant Bit Rate* (CBR): Utilizando Constant Bit Rate (ou taxa de bits constante) o tráfego gerado pela aplicação não possui variações. Por exemplo, o padrão de compactação MPEG1 comprime vídeo em uma taxa constante de bits, sendo que para vídeos de baixa qualidade a taxa é de 1.15 Mbps e para vídeo em boa qualidade a taxa é de 3 Mbps.
- *Variable Bit Rate* (VBR): Utilizando Variable Bit Rate (ou taxa de bits variável) o tráfego gerado pela aplicação possui variações, por exemplo, em um vídeo em que as imagens se movem rapidamente a taxa é maior que quando há pouca movimentação. O padrão de compactação MPEG2 é uma família que suporta taxas variáveis de compressão (ROSE, 1995).
- Seqüência de Mensagens: Algumas aplicações em uma rede são implementadas através de processos que trocam mensagens. Estas mensagens são de tamanho variável e a ocorrência de mensagens pode ser em qualquer intervalo de tempo dependendo da aplicação (não há previsão de quando as mensagens irão ocorrer).

Levando-se em consideração que o tráfego gerado pelas aplicações pode assumir as diversas formas destacadas acima, prover Qualidade de Serviço significa atender aos requisitos exigidos pela aplicação para a transmissão de determinado tráfego. No entanto, em situações práticas, algumas vezes é inviável prover o máximo de QoS para todas as aplicações. Com isto, existem vários níveis de QoS que podem ser criados como mostra a Figura 8.

Segundo Walrand e Varaiya (2000), utilizando variados níveis de QoS é possível priorizar serviços que exigem muitos recursos em tempo real, como por exemplo serviços de Vídeo sobre Demanda (VoD - *Video on Demand*) e realizar a entrega de outras informações não prioritárias de maneira a não sobrecarregar a rede.

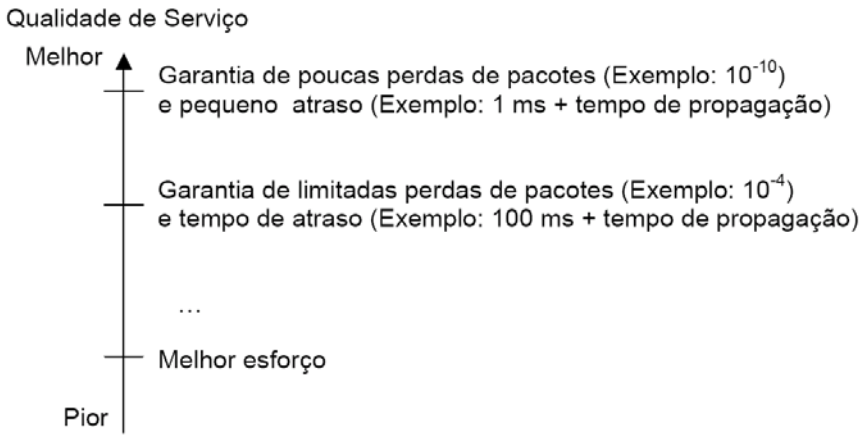

Figura 8 – Níveis de Qualidade de Serviço (WALRAND e VARAIYA, 2000)

3.4 – Alguns Sistemas de QoS

Esta seção apresenta dois trabalhos sobre QoS que possuem enfoque sobre a segurança.

3.4.1 – Middleware de Comunicação

Um trabalho que apresenta a implementação de QoS em conjunto com alguns aspectos de segurança foi apresentado por Guelfi (2002). Neste trabalho, foi realizada a proposta de uma *middleware*, ou camada intermediária, de comunicação para prover o gerenciamento de serviços multimídia (GUELFI, 2002).

Esta *middleware* foi desenvolvida para suportar aplicações multimídia distribuídas e atender de maneira mais completa os critérios de eficiência, utilidade, modularidade e heterogeneidade (GUELFI, 2002).

Desta forma, a arquitetura funcional da *Middleware* foi definida entre a API (*Application Programming Interface*) e o nível de acesso à rede e protocolos (GUELFI, 2002). Isto permite que diversas aplicações interajam com a *Middleware* através da API e também que sejam utilizadas diversas tecnologias de rede e protocolos, uma vez que a *Middleware* se encontra acima da camada de acesso à rede e protocolos.

A Figura 9 apresenta a arquitetura funcional da *Middleware* proposta por Guelfi (2002). Nesta figura podem ser visualizados os módulos de gerenciamento da *Middleware* que são responsáveis por diversas funções como: segurança, QoS, sessões de usuário, etc.

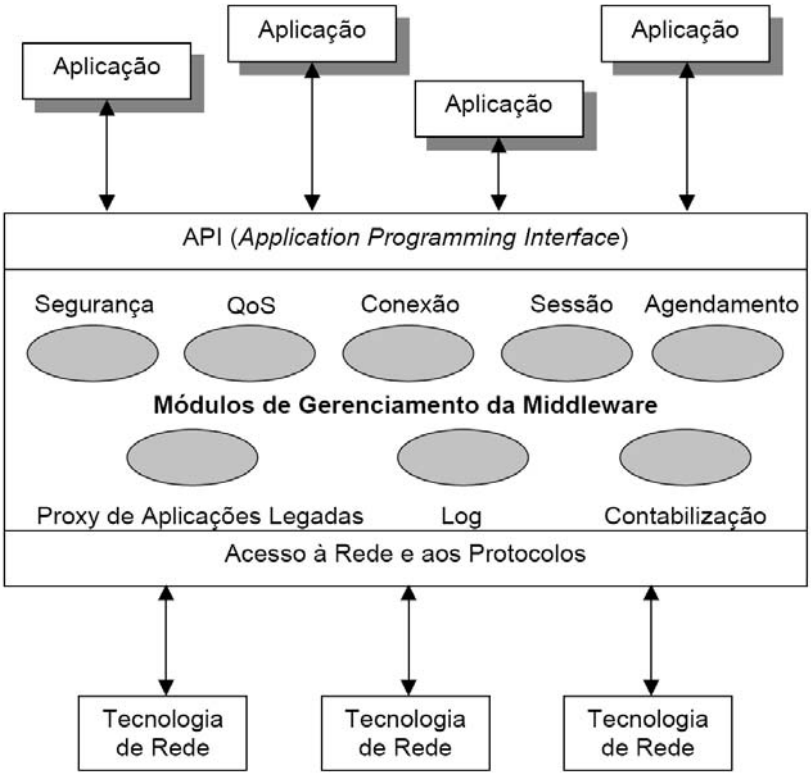

Figura 9 – Arquitetura Funcional da Middleware apresentada por Guelfi (2002)

A seguir, são descritas as características dos principais módulos apresentados na Figura 9 (GUELFI, 2002):

- O módulo de Gerenciamento de **Segurança** é responsável por controlar os níveis de acesso dos usuários à *Middleware* e prover os serviços básicos de segurança. Desta forma, ele deve estar relacionado com algumas características de PKI (*Public Key Infrastructure* – Infra-estrutura de Chave Pública);
- O Módulo de Gerenciamento da **QoS** é o responsável por tratar das questões referentes à disponibilidade de recursos (largura de banda, Tempo de CPU, tempo de atraso, etc.);
- O Módulo de Gerenciamento da **Conexão** desempenha funções inerentes aos protocolos e tecnologias de rede suportadas pela *Middleware* como o estabelecimento de comunicações ponto a ponto ou *multicast*, encerramento de conexões ativas, etc.;
- O Módulo de Gerenciamento de **Sessão** Multimídia realiza operações de controle referentes aos aspectos de uma sessão multimídia estabelecida. Este módulo desempenha um papel de entidade de gerenciamento central;
- O Módulo de Gerenciamento Dedicado ao **Agendamento** de Sessões Multimídia é responsável pelo agendamento prévio dos serviços oferecidos pela *Middleware*;
- O Módulo de Gerenciamento **"Proxy de Aplicações Legadas"** é encarregado de atender a heterogeneidade característica dos ambientes de rede disponíveis;

- O Módulo de Gerenciamento de **Logs** é responsável por registrar os itens mais importantes das atividades realizadas por cada aplicação de usuário estabelecida através da *Middleware*;
- O Módulo de Gerenciamento de **Contabilização** é encarregado de controlar o custo associado à utilização de recursos por parte de uma aplicação.

Para entender o funcionamento da *Middleware*, a Figura 10 apresenta um fluxograma geral de operação dos componentes da *Middleware*. Nesta figura pode-se observar que assim que os parâmetros requisitados pelo usuário são abordados pelo Gerente de Sessão, este repassa-os ao Gerente de QoS quando a sessão é aceita (passos 1 e 2 da Figura 10) (GUELFI, 2002).

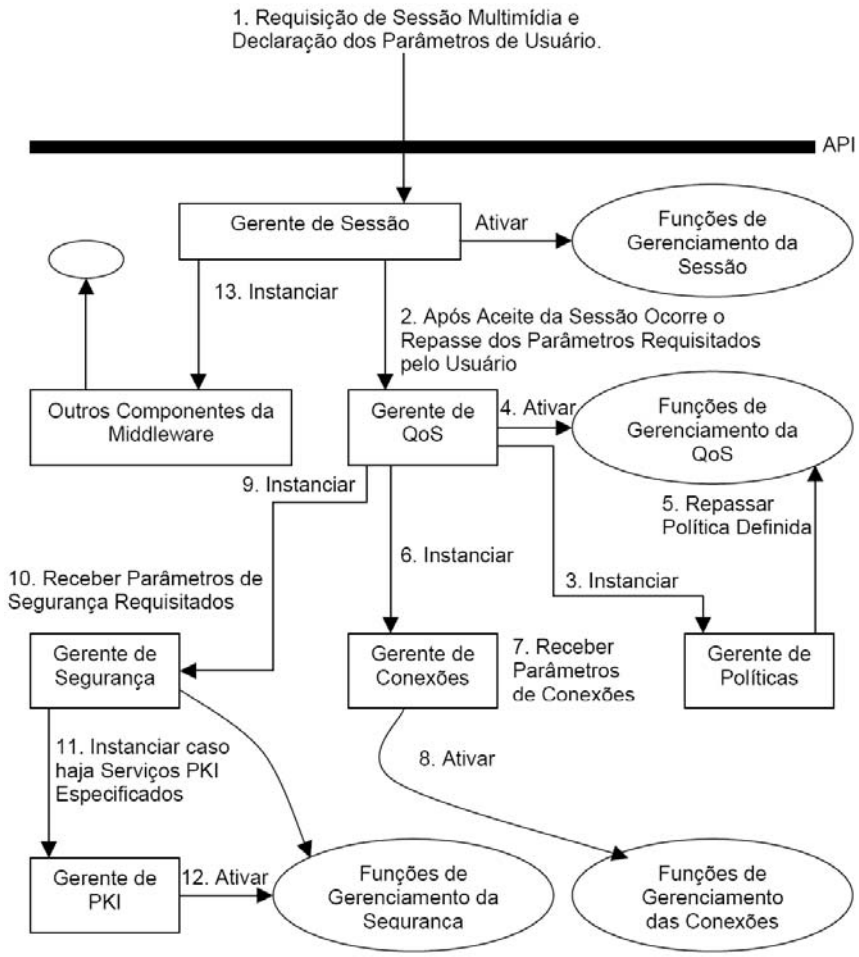

Figura 10 – Fluxograma Geral de Operação dos Componentes da Middleware (GUELFI, 2002)

O Gerente de QoS verifica os recursos disponíveis, ativa as funções do gerenciamento de QoS e provê a política de adaptação vigente para aquela sessão (passos 3, 4 e 5 da Figura 10). Após estas operações, o gerente de QoS troca os parâmetros compatíveis com o Gerente de Conexões ativando, se possível, suas funções de gerenciamento (passos 6, 7 e 8 da Figura 10). Após isto, o Gerente de QoS ainda é o responsável por habilitar as funções do Gerenciamento de Segurança (passos 9, 10, 11 e 12 da Figura 10).

Finalmente, o Gerente de Sessão pode interagir com os demais componentes da *Middleware* (passo 13 da Figura 10).

3.4.2 – CriptoQoS

Outro trabalho que implementa QoS em conjunto com segurança é a middleware CriptoQoS proposta por Meylan (2003). A CriptoQoS tem como objetivo atender os requisitos de aplicações distribuídas fornecendo QoS e segurança.

A arquitetura da CriptoQoS é formada por: Agente, CriptoQoS Proxy, Gerente de Políticas, Gerente de QoS, Gerente de Segurança, Gerente de Sessão e Monitor de Falhas como mostra a Figura 11 (MEYLAN, 2003).

O Agente é responsável por coletar informações sobre os recursos computacionais disponíveis e solicitar autorização e parâmetros de QoS e segurança para requisições de conexão.

O CriptoQoS Proxy é um *proxy* de interface entre os Agentes e os servidores que não utilizam a API CriptoQoS.

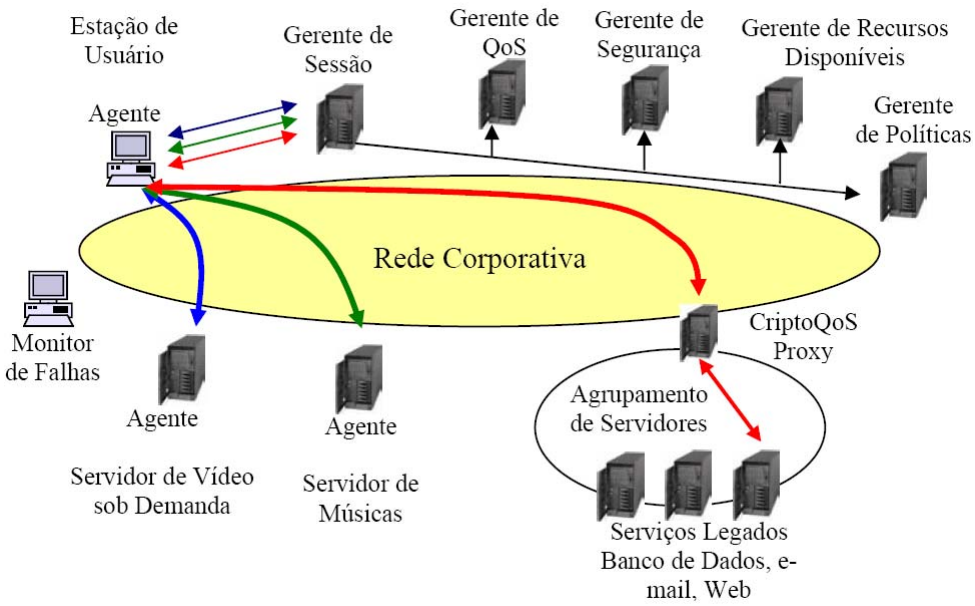

Figura 11 – Arquitetura da CriptoQoS (MEYLAN, 2003)

Tabela 16 – Propriedades definidas pelo Gerente de Políticas (MEYLAN, 2003)

Propriedade	Valores Possíveis
Nome da Política	String alfanumérica
Autenticação	• Sem autenticação • Fraca (login + senha) • Média (cartão + senha) • Forte (cartão + biometria)
Segurança (Confidencialidade)	• Sem confidencialidade • Fraca (criptografia fraca) • Média (criptografia forte) • Forte (criptografia forte + assinatura de mensagens)
Garantia de Disponibilidade de Recursos de Rede (QoS)	• Sem garantia (melhor esforço) • Fraca (banda mínima garantida) • Média (banda constante garantida) • Forte (banda constante e atraso máximo garantidos)

Uma característica importante na CriptoQoS é a presença de um Gerente de Políticas, que é responsável por definir políticas de utilização dos serviços disponíveis na rede. As

políticas são definidas através de uma linguagem de alto nível e uma mesma política pode definir o acesso a diversos recursos da rede.

A Tabela 16 apresenta as propriedades que podem ser definidas pelo Gerente de Políticas e a Tabela 17 mostra um conjunto de exemplos de políticas que podem ser definidas.

Tabela 17 – Exemplos de Políticas (MEYLAN, 2003)

Política: Serviços Convencionais	
Autenticação	Nenhuma
Confidencialidade	Nenhuma
QoS	Melhor Esforço

Política: Vídeos de Treinamento	
Autenticação	Fraca
Confidencialidade	Fraca
QoS	Forte

Política: e-mail Corporativo	
Autenticação	Fraca
Confidencialidade	Fraca
QoS	Melhor Esforço

Política: Acesso Remoto (Internet)	
Autenticação	Média
Confidencialidade	Forte
QoS	Melhor Esforço

Política: Servidores Restritos	
Autenticação	Forte
Confidencialidade	Forte
QoS	Fraca

Política: Reuniões Remotas (Rede Corporativa)	
Autenticação	Fraca
Confidencialidade	Fraca
QoS	Média

3.5 – Suporte à Segurança Oferecido pelos Sistemas de QoS

Como descrito na seção anterior, existem vários projetos que objetivam estabelecer conexões seguras, no entanto há uma grande dificuldade para a escolha dos algoritmos a serem utilizados uma vez que existem muitos algoritmos que variam em nível de segurança, objetivo e desempenho. Com isto existem alguns projetos que consideram parâmetros para a escolha dos algoritmos/chaves a serem utilizados nas conexões.

Nesta seção são novamente examinados os sistemas apresentados por Guelfi (2002) e Meylan (2003) com enfoque na segurança oferecida por eles e como ela é alcançada.

No projeto proposto por Guelfi (2002) é apresentada uma Middleware de comunicação para o gerenciamento de serviços multimídia a qual é composta por vários módulos para prover QoS e segurança em uma transmissão multimídia.

Nesta *Middleware*, o módulo responsável pela segurança é chamado de Gerente de Segurança e tem a função de prover serviços como autenticação, não-repúdio, confidencialidade e integridade. O módulo Gerente de Segurança possui um componente chamado de Gerente de PKI para o controle de certificados digitais.

Segundo Guelfi (2002), o nível de segurança provido por esse módulo deve ser variável e dependente da demanda de usuários e do valor da informação. No momento em que uma conexão estiver sendo estabelecida e for verificado que os parâmetros de segurança fornecidos pelo usuário informam a necessidade de criptografia, será feita uma análise se os parâmetros solicitados (por exemplo, algoritmo escolhido e tamanho de chave) são viáveis de serem utilizados naquela conexão. O autor sugere que esta análise seja feita com base em parâmetros como largura de banda, tempo de atraso e utilização da CPU.

O trabalho apresentado por Meylan (2003) também apresenta uma integração entre QoS e segurança. Uma característica da CriptoQoS é a presença de elementos chamados de agentes que capturam informações sobre os recursos computacionais disponíveis nas estações. Essas informações são importantes, pois quando um novo pedido de conexão for realizado elas deverão ser consultadas para autorizar ou não a nova conexão.

Para a escolha do algoritmo de criptografia a ser utilizado, bem como o tamanho da chave, existe um elemento chamado Gerente de Políticas. Este elemento estabelece as regras para segurança (autenticação e confidencialidade) e os parâmetros de QoS requeridos.

No entanto, os serviços de segurança são providos por um elemento chamado Gerente de Segurança. Este elemento é responsável em fornecer autorização para o

estabelecimento de uma nova conexão e em implementar os serviços de criptografia e certificação digital.

Quando uma conexão for solicitada, o Gerente de Segurança deverá identificar os parceiros da comunicação. Estas informações são então repassadas ao Gerente de Políticas que definirá se a conexão será aceita e quais os parâmetros necessários para ela. De acordo com a política estabelecida pelo Gerente de Políticas, o Gerente de Segurança realizará a escolha dos algoritmos, tamanho das chaves e mecanismos de autenticação, porém essa seleção é realizada pelo usuário de uma forma estática (antes de realizar a conexão).

Estes projetos sugerem e exemplificam a necessidade de se ter um sistema que consiga acompanhar, em tempo real, as diferentes características estabelecidas entre as conexões, porém não chegam à fase de implementação deste acompanhamento em nível de segurança, mas sim em nível de QoS. Interessante notar que estes projetos utilizaram poucos algoritmos e não consideraram a tecnologia física de rede e protocolos.

Os sistemas apresentados por Guelfi e Meylan são sistemas voltados para a qualidade de serviço que também visam manter a segurança do sistema, no entanto, nestes sistemas é considerado um único algoritmo de criptografia. Seria interessante que esses sistemas tivesse outras implementações e realizassem adaptações também quanto ao algoritmo utilizado em determinada situação. O SICO possui a implementação de vários algoritmos de criptografia e permite uma adequação do melhor algoritmo para determinada situação e a troca dinâmica de algoritmos criptográficos e chaves, no entanto não são considerados aspectos de qualidade de serviço.

CAPÍTULO 4 – DESCRIÇÃO INICIAL DO SICO

Este capítulo apresenta uma descrição inicial do Sistema Inteligente de Comunicação, que é chamado, neste trabalho, de SICO.

Como descrito no início do texto, existem vários algoritmos criptográficos, cada um com seus objetivos e características, o que torna muito difícil a escolha de um algoritmo ou um conjunto de algoritmos mais adequado para uma determinada conexão, visto que as condições de uma conexão podem variar muito rapidamente, pois pode haver congestionamento da rede, sobrecarga de processamento em uma das estações e outros fatores que influenciam na escolha dos algoritmos.

Com isso, foi proposto um sistema que consiga identificar os principais fatores relacionados com esta escolha, e desta forma determinar um melhor conjunto de algoritmos para realizar uma determinada conexão e estabelecer a troca de informações de maneira segura e eficiente.

Inicialmente foi proposto um protótipo do SICO em linguagem C (CHIARAMONTE, 2003). Neste protótipo existe um módulo responsável pela comunicação utilizando *Sockets*. A implementação desse protótipo inicial permitiu que uma máquina somente enviasse dados cifrados e a outra recebesse esses dados e realizasse a decifragem.

O módulo mais importante para o sistema é o módulo responsável em avaliar as condições do ambiente e definir, através de um conjunto de parâmetros preestabelecidos, qual o melhor algoritmo criptográfico a ser utilizado. Estes parâmetros são listados na seção 4.2. Na arquitetura inicial descrita em 4.1, este módulo é chamado de Gerente.

A seguir a arquitetura do sistema é definida em mais detalhes e é apresentado um primeiro protótipo utilizando a linguagem C.

4.1 – Estrutura Inicial do Protótipo em C

Para uma implementação inicial de um protótipo em linguagem C em que os dados cifrados trafegam em uma única direção, o sistema contém dois módulos principais: um para a criptografia e envio de mensagens criptografadas chamado *Sender*; e um outro módulo para receber as mensagens criptografadas e decriptografá-las, chamado *Receiver*.

O módulo *Sender* é composto por 4 (quatro) *threads*, sendo que 3 (três) dessas *threads* trabalham no modelo *pipeline* e a comunicação interna é feita por buffers compartilhados.

Cada uma das *threads* possui uma função específica:

- A ***Thread* Receptor** abre uma conexão com um cliente utilizando *Sockets* TCP e fica à espera de dados para criptografar. Assim que os dados chegam, a *Thread* coloca-os no buffer de recepção para serem criptografados.
- A ***Thread* Cripto** retira os dados do buffer de recepção e realiza a criptografia. Após a criptografia os dados são colocados em um buffer de emissão.
- Finalmente a ***Thread* Emissor** retira os dados do buffer de emissão e os envia através de uma conexão utilizando *Sockets* TCP.
- A ***Thread* Gerente** verifica periodicamente o nível dos buffers compartilhados para detectar alguns parâmetros como velocidade de recepção, velocidade de criptografia e velocidade de envio. Esta *Thread* é a responsável por definir a necessidade da troca do algoritmo ou chave utilizada.

O módulo *Receiver* possui uma estrutura parecida. Existem duas t*hreads* que trabalham em *pipeline* e uma *thread* como um controle. A seguir as funções destas *threads* são descritas com maior detalhes:

- A ***Thread* Receptor** deste módulo tem a mesma função da *Thread* Receptor do módulo anterior, armazenando os dados que chegam em um buffer compartilhado.
- A ***Thread* Decripto** realiza a decriptografia dos dados e os envia para uma saída a ser definida (pode ser arquivo ou rede).
- A função da ***Thread* de Controle** é realizar verificações periódicas no nível do buffer e enviá-las periodicamente para o módulo Emissor. Essa verificação pode ser utilizada como um parâmetro de comparação de funcionamento dos dois módulos e comprovar as medições no buffer realizadas pelo módulo Emissor.

A Figura 12 mostra a arquitetura geral do sistema proposto (SICO). Nela pode ser observado o fluxo de informações entre os módulos e o fluxo interno entre as *threads*.

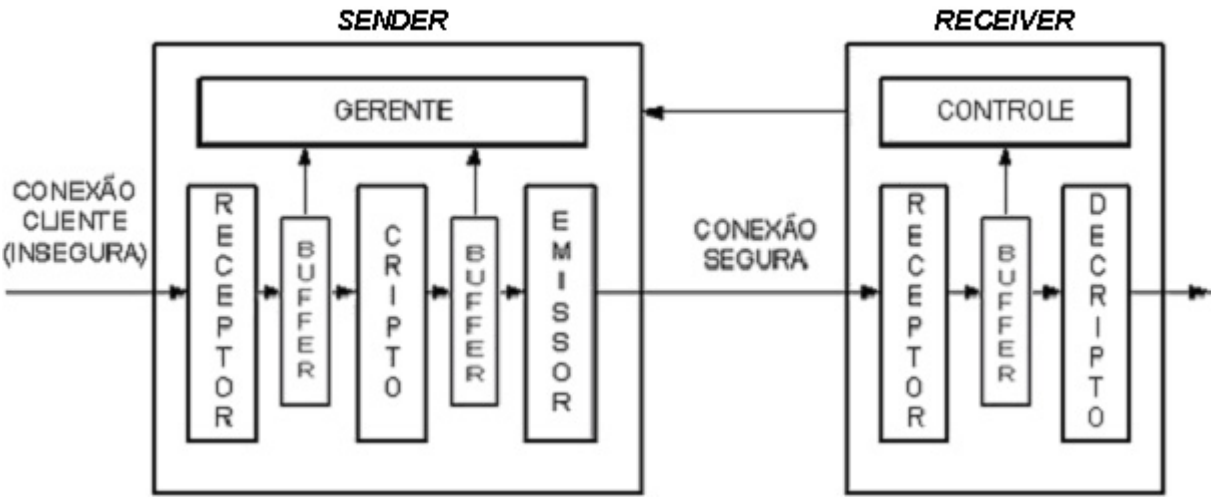

Figura 12 – Arquitetura Inicial do Sistema SICO

Esta estrutura apresentada foi utilizada como base em um primeiro protótipo desenvolvido em linguagem C. Alguns dados sobre este primeiro protótipo são apresentados nas seções 4.2, 4.3.

4.2 – Parâmetros Considerados e Algoritmos de Decisão

Neste primeiro protótipo implementado em linguagem C foram considerados parâmetros referentes ao uso de processador nas máquinas envolvidas, velocidade da rede utilizada e o nível de segurança desejado pelo usuário.

Para capturar estes parâmetros foram utilizados os buffers do emissor descritos na seção 4.1 (buffer de recepção e buffer de emissão). Entretanto, novas formas de medidas destes parâmetros, bem como novos parâmetros que possam ser considerados foram estudados para a definição da arquitetura completa apresentada no capítulo 5.

Com estes dois buffers para dados são possíveis quatro situações diferentes que são analisadas a seguir:

- **Buffer de recepção baixo e buffer de emissão baixo**: Os dados para serem criptografados estão chegando a uma velocidade inferior à velocidade do processador. Neste caso é possível, caso o nível de segurança desejado pelo usuário exija, utilizar uma chave mais forte na criptografia. No entanto esta mudança não é prioritária nem corretiva, uma vez que os processos de criptografia e decriptografia estão ocorrendo na mesma velocidade.
- **Buffer de recepção baixo e buffer de emissão alto**: Os dados estão sendo criptografados mais rápido do que podem ser enviados. Isto pode ocorrer por

um congestionamento na rede de saída ou por sobrecarga na máquina destino ao realizar a decriptografia. Neste caso, deve-se avaliar a situação para tomar uma decisão, uma vez que, se o problema for congestionamento, seria interessante utilizar uma chave mais forte na criptografia (o que exige mais tempo e aliviaria o congestionamento da rede). No entanto, se o problema for sobrecarga de processamento na estação destino, utilizar uma chave mais forte iria piorar o problema, pois mais tempo deveria ser gasto na decriptografia. Importante lembrar que sempre os ajustes serão de acordo com o nível de segurança desejado pelo usuário.

- **Buffer de recepção alto e buffer de emissão baixo**: Significa que os dados estão chegando mais rápido do que podem ser criptografados. Neste caso, uma medida possível seria utilizar um nível de criptografia mais fraco, caso se enquadre no nível de segurança desejado pelo usuário.
- **Buffer de recepção alto e buffer de emissão alto**: Duas condições podem levar a esta situação: **A primeira** seria que a rede está funcionando sem congestionamentos e todos os processos (criptografia e decriptografia) estão utilizando todo o poder de processamento possível. Neste caso, se necessário seria possível utilizar um nível de criptografia mais fraco para aliviar um pouco o processador. Isso seria conseguido através da mudança do algoritmo ou do tamanho da chave, o que pode significar uma perda no nível de segurança. **A segunda situação** seria após algum período com o buffer de emissão cheio (por problema na rede ou velocidade de processamento na decriptografia que não pode ser resolvido) a *thread* de criptografia não teria mais espaço para depositar os dados criptografados, sendo obrigada a reduzir a velocidade, o que faria com que os dados que chegam para ser

criptografados tenham que aguardar no buffer de recepção. Nesta segunda situação, não há o que se fazer. Uma opção seria, em trabalhos futuros, implementar uma arquitetura mais completa que permita a replicação do serviço para que este continue ativo em outras aplicações funcionando em outras máquinas.

4.3 – Exemplo de Funcionamento do Protótipo em C

Como descrito anteriormente, foi desenvolvido um primeiro protótipo do sistema SICO utilizando a linguagem C. Esta implementação opera inicialmente com um único algoritmo criptográfico trocando as chaves de acordo com a necessidade.

O algoritmo criptográfico utilizado neste protótipo é uma versão do algoritmo Posicional que trabalha com blocos de 32 bits. Mais detalhes sobre esta versão do algoritmo Posicional podem ser encontrados em (MORENO et. al., 2005)(CHIARAMONTE e MORENO, 2005).

Foram definidos 3 níveis de segurança:

- nível 0: o grau da chave varia entre 7 e 21;
- nível 1: o grau da chave varia entre 22 e 36;
- nível 2: o grau da chave varia entre 37 e 51.

Para realizar o controle dos buffers compartilhados foram utilizados os mecanismos de Semáforos e Seção Crítica (TANENBAUM, 2000).

O módulo emissor recebe conexões através da porta 9000 e envia as informações criptografadas através da porta 9001 (estas portas foram escolhidas aleatoriamente). O módulo receptor recebe as informações através da porta 9001 e realiza a decriptografia.

Os resultados iniciais obtidos comprovam que é viável utilizar este tipo de sistema de reconfiguração dinâmica de chaves, uma vez que o desempenho deste sistema, utilizando o nível máximo de segurança (nível 2) se mostrou apenas um pouco mais lento que utilizar chaves de grau 51 com o mesmo algoritmo, o que significa que o *overhead* causado pela *Thread* Gerente e pela troca das chaves não é muito significativo. Apesar de pequeno, o impacto deste *overhead* tende a diminuir com o uso de outros algoritmos criptográficos, uma vez que as variações de desempenho entre algoritmos diferentes são grandes, o que justifica o uso deste sistema.

A Figura 13 e na Figura 14 mostram telas de execução do sistema. Na Figura 13 pode-se perceber que o grau da chave utilizada está sendo reduzido como uma medida para tentar reduzir o nível do buffer de recepção, que está entre 99% e 100% ocupado. Esta redução do grau da chave continuará até que o nível do buffer encontre um limite considerado satisfatório ou o grau da chave chegue ao seu limite mínimo dentro do nível de segurança desejado.

Na Figura 13 e na Figura 14 pode-se observar que quando ocorre a mudança da chave no módulo emissor, ela é repassada ao receptor, como mostra o trecho destacado pela elipse referente à troca de chave.

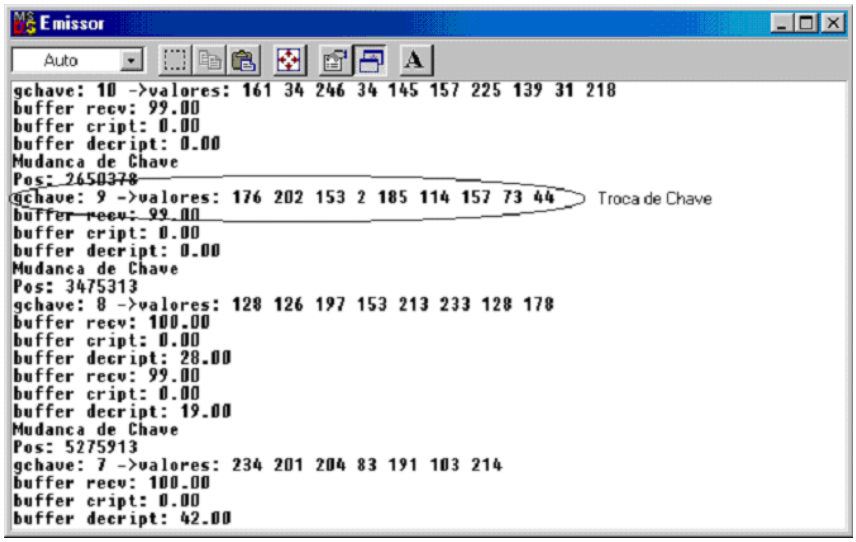

Figura 13 – Exemplo do Módulo Emissor (CHIARAMONTE e MORENO, 2005)

Na Figura 14 existe também um destaque referente à redução do nível do buffer de recepção. Quando o grau da chave é alterado para 7 o nível do buffer cai de 31% para 9%. Isto ocorre pois a velocidade de decriptação aumenta significativamente com a redução da chave, ultrapassando 1 Mbyte/segundo.

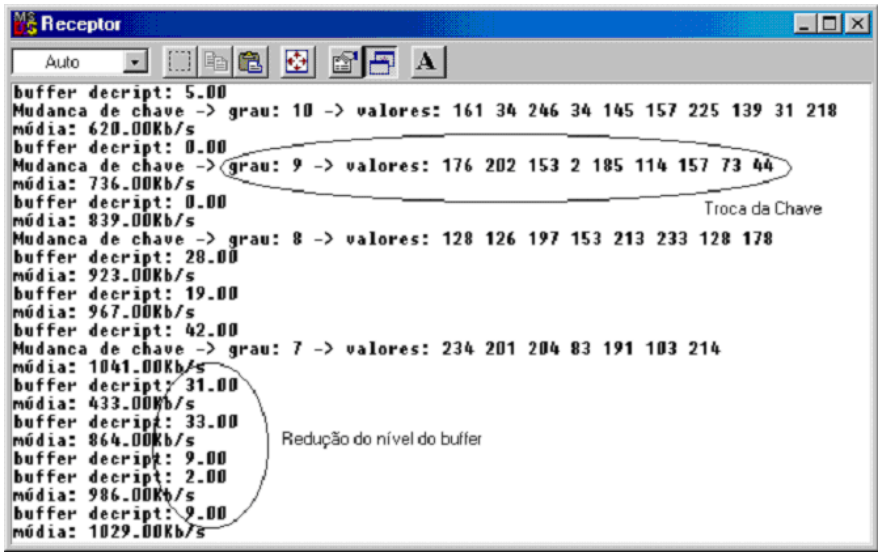

Figura 14 – Exemplo do Módulo Receptor (CHIARAMONTE e MORENO, 2005)

CAPÍTULO 5 – ARQUITETURA COMPLETA DO SICO

A partir da implementação do primeiro protótipo em linguagem C foi possível verificar as vantagens e desvantagens relacionadas com a linguagem utilizada para este tipo de sistema. A implementação em linguagem C apresentou um bom desempenho e atendeu aos requisitos iniciais propostos, sendo isso comprovado através dos testes realizados e apresentados no capítulo anterior. No entanto, houve grandes dificuldades quanto à criação e sincronização de *threads* fazendo com que o sistema ficasse restrito a um único sistema operacional, no caso o Windows 98. Outro problema, também relacionado com o uso de múltiplas *threads*, é a complexidade para controlar e sincronizá-las para o sistema completo, em que a comunicação ocorre nas duas direções utilizando os recursos disponíveis pela linguagem C.

Desta forma, para a implementação da arquitetura completa, optou-se por utilizar a linguagem Java que possui recursos que facilitam a criação e controle das *threads* e não exige chamadas diretas ao sistema operacional o que impossibilitaria a migração do SICO para diferentes sistemas operacionais.

Outra vantagem apresentada pela linguagem Java é a programação orientada a objetos, que facilita a estruturação do sistema como um todo e viabiliza a adição de novos recursos ao sistema sem grandes alterações.

Outra linguagem que poderia ser utilizada é a linguagem C++ que também oferece recursos de orientação a objetos. Entretanto seria necessária a elaboração de um controle para a sincronização das *threads*. Desta forma optou-se por utilizar a linguagem Java utilizando JNI para as partes críticas que exigem mais desempenho.

Seguindo a abordagem orientada a objetos também é possível definir uma API em Java que permita que sejam desenvolvidas aplicações utilizando o SICO, os detalhes de implementação se encontram na seção 5.3.

5.1 – Estrutura da Implementação em Java

Inicialmente foram estudadas quais as principais classes necessárias para melhor organizar o uso dos algoritmos de criptografia no SICO, uma vez que uma característica principal do sistema é alternar os algoritmos dinamicamente, exigindo assim que as classes que implementam estes algoritmos possuam métodos padrões. Adicionalmente, foi construída uma ferramenta para integrar vários algoritmos criptográficos e validar a estrutura de classes criada, esta ferramenta foi chamada de WEBCry (MORENO et. al. 2005).

Figura 15 – Interface da Ferramenta WEBCry

A Figura 15 apresenta a interface da ferramenta WEBCry que permite a escolha de vários algoritmos criptográficos de diferentes tipos e, após a definição da chave, realizar operações de cifragem e decifragem sobre textos ou arquivos.

Para uma melhor organização, os algoritmos foram agrupados de acordo com o seu tipo e objetivo através de *Interfaces* em Java. Segundo Sun (2005) uma *Interface* é um conjunto de definições de métodos e valores constantes que devem ser implementados por classes que definem esta *Interface*.

Com isto, foram definidas *Interfaces* em que constam os principais métodos referentes a cada tipo de algoritmo. Quando um algoritmo for implementado ele deverá definir qual *Interface* ele irá implementar, de acordo com seu tipo. Isto obrigará todos os algoritmos do mesmo tipo a ter os mesmos métodos de acesso, facilitando a sua utilização no SICO.

A seguir são apresentadas três *Interfaces* básicas:

A *Interface* Algoritmos possui os métodos mais genéricos que são comuns entre os algoritmos criptográficos utilizados, sendo eles: *doFinal* responsável por finalizar uma operação envolvendo criptografia; *update* responsável por realizar uma operação parcial de criptografia; e *getOutputSize* que retorna qual o tamanho necessário para armazenar o resultado após uma chamada ao método *doFinal*.

A *Interface* Simétrico foi criada para agrupar os algoritmos simétricos de criptografia (Exemplo: AES, DES, IDEA). Ela é composta dos métodos: *defineChave* que é responsável por definir a chave que será utilizada na operação; *criaChave* responsável por criar uma chave aleatória; *obtemChave* que retorna qual a chave atualmente utilizada; *defineModo* para definir se será realizada a criptografia ou decriptografia; *limpaChave* responsável por apagar a chave atualmente em uso; e *temChave* para verificar se existe alguma chave definida para algoritmo.

A *Interface* Assimétrico, criada para os algoritmos assimétricos é semelhante à utilizada para os algoritmos simétricos, no entanto, existe uma grande diferença no método de definição de chaves uma vez que devem existir métodos específicos para manipulação da chave.

A partir desta estrutura foi possível criar um pacote (SICO.Cripto) com a implementação dos algoritmos de criptografia utilizados no SICO. O diagrama de classes desta estrutura aparece na Figura 16. Esta estrutura permite acrescentar novos algoritmos ao SICO apenas seguindo a interface apropriada.

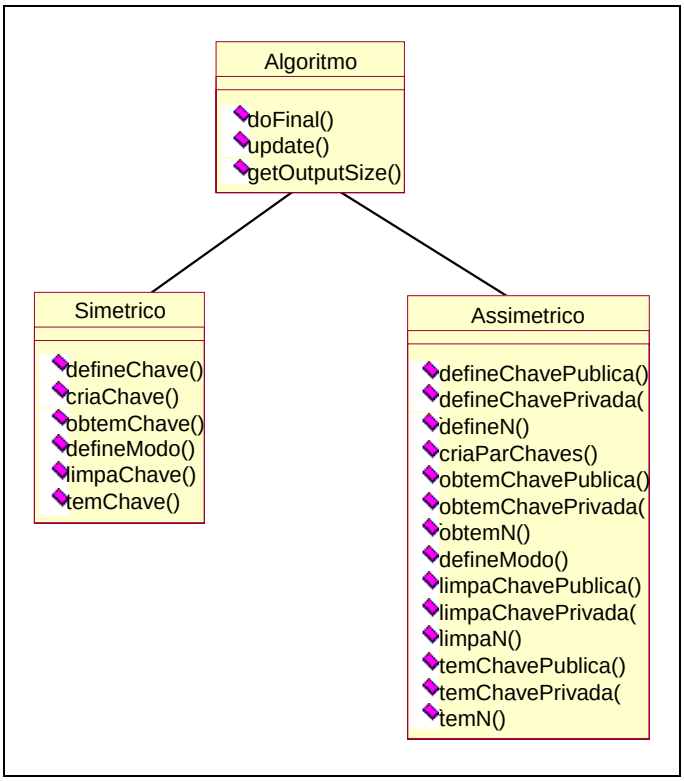

Figura 16 – Diagrama de classes do pacote Cripto

Como descrito anteriormente, esta implementação em Java deverá realizar a comunicação nas duas direções. Desta forma, o sistema foi dividido em duas partes: cliente e

servidor. Inicialmente o cliente requisita ao servidor uma conexão utilizando um determinado nível de segurança; o servidor então atende a requisição e inicia-se a comunicação segura utilizando **um protocolo que permite a troca de chaves e algoritmos**.

Para isto foi necessário reestruturar a arquitetura proposta no Capítulo 4. Nesta nova arquitetura é necessário um módulo específico responsável pelo protocolo de comunicação, pois é necessário o envio de sinais de controle que indicam troca de chave ou algoritmo. Também são necessários módulos específicos para o controle de decisão sobre a troca de algoritmo e a troca de chaves de maneira segura.

A Figura 17 apresenta a arquitetura geral do SICO. Através dela pode-se visualizar os módulos Protocolo, Key Exchange (para realizar trocas de chaves de maneira segura) e Controle de Decisão. A partir desta arquitetura foi possível definir o diagrama de classes do sistema sendo que para a implementação do protocolo foram utilizadas as classes RecebeSeguro e ConexaoSegura; para o controle de decisão foi utilizada a classe Verifica; e para a troca de chaves de modo seguro foi definida a classe KeyExchange. A Figura 18 apresenta o diagrama de classes utilizado na implementação.

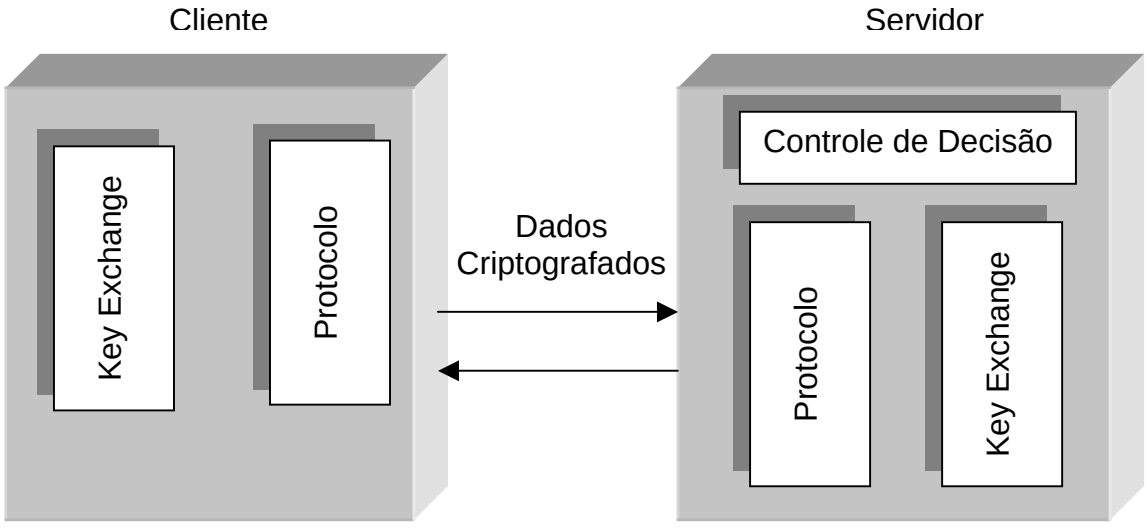

Figura 17 – Esquema Geral do SICO

Para o funcionamento da arquitetura completa também foi necessário definir um protocolo que permitisse passar informações relativas à chave e algoritmos utilizados. Com isso foi possível implementar a troca de chaves e algoritmos dinamicamente, entretanto, inicialmente essa troca de chaves ocorre sem um algoritmo específico para distribuição de chaves, o que diminui significativamente a segurança do sistema.

Como descrito anteriormente, para manipular os dados relativos ao protocolo foi necessária a implementação das classes RecebeSeguro e ConexaoSegura. Essas classes são responsáveis por interpretar os dados existentes no cabeçalho e realizar as ações apropriadas, bem como *bufferizar* as informações quando necessário.

Nesta nova arquitetura existe apenas um buffer responsável por armazenar os dados que chegam da rede. Este buffer é utilizado pela classe RecebeSeguro para auxiliar no processamento do protocolo. Importante enfatizar que após alguns testes foi possível detectar a importância do tamanho deste buffer para o correto funcionamento e bom desempenho do sistema. Esses dados são apresentados na seção 5.4.

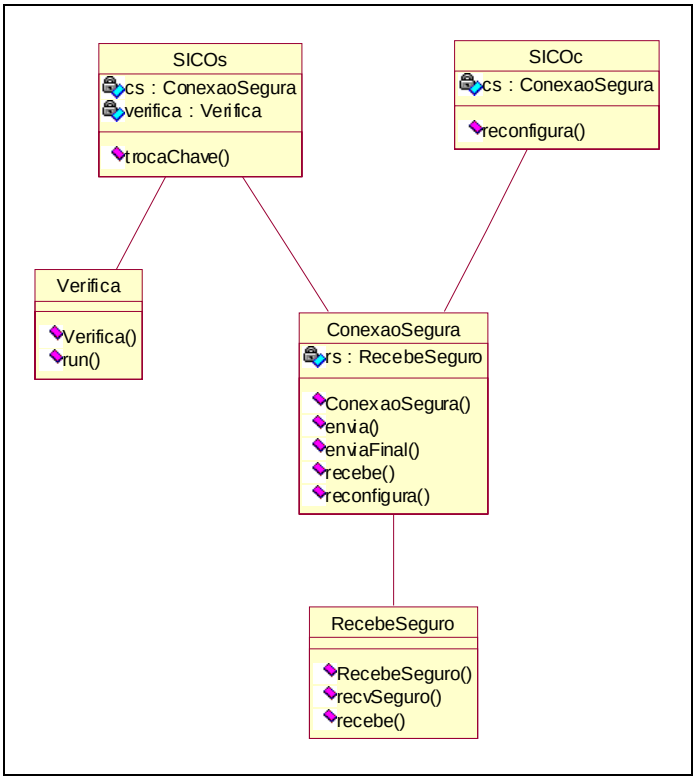

Figura 18 – Diagrama de classes do SICO

5.2 – Algoritmos utilizados no SICO

Como descrito no Capítulo 2, foram realizados testes de desempenho entre as implementações e verificaram-se quais obtiveram melhores resultados. Com isso, foram escolhidas as implementações do DES e do AES presentes na JCA e as implementações do RC6, Serpent, Twofish, Mars e AES desenvolvidas por (GLADMAN, 2006), utilizando JNI.

Para diferenciar entre as duas versões presentes do algoritmo AES, a primeira versão (utilizando JCA) será denominada de AES; e a segunda versão (GLADMAN, 2006) será denominada JAES.

Como os algoritmos implementados em linguagem C foram inseridos utilizando JNI não houve a necessidade de reimplementação dos algoritmos em linguagem Java.

5.3 – Protocolo para a troca de chaves e algoritmos

Para permitir a troca de chaves e algoritmos em tempo de execução do sistema, foi elaborado um protocolo de comunicação em que troca ocorra de maneira sincronizada entre as partes envolvidas.

Com isso, a estrutura do pacote utilizado pelo sistema é apresentada na Tabela 18. Cada célula da tabela representa um byte. O byte 1 é uma informação de controle conforme detalhado na Tabela 19; os bytes 2 e 3 contêm a informação referente ao tamanho da mensagem e os bytes seguintes são os dados da mensagem.

Tabela 18 – Estrutura do pacote

1	2	3	4	...	n
Controle	**Tamanho**	**Tamanho**	**Dados**	**...**	**Dados**

Pode-se observar na Tabela 19 as informações contidas no byte de controle. O bit menos significativo (bit 1) informa se a mensagem é referente a uma nova chave ou a dados. O segundo bit é utilizado por parte do protocolo de troca de chaves para uma confirmação de que a chave foi trocada em uma das partes. Os outros três bits seguintes informam qual o algoritmo que deverá ser utilizado, caso o bit 1 indique que a mensagem é para troca de chave.

Tabela 19 – Estrutura do byte de controle

8	7	6	5	4	3	2	1
reservado	**reservado**	**reservado**	**algoritmo**	**algoritmo**	**algoritmo**	**chave_ack**	**chave**

Desta forma, quando é necessário ocorrer uma troca de chave ou algoritmo, o servidor gera o novo conjunto e envia ao cliente um pacote com o bit 1 do byte de controle igual a ‘1’; os bits 3, 4 e 5 do byte de controle indicam o algoritmo (ver Tabela 20). Os bytes referentes ao tamanho do pacote contêm o tamanho da chave utilizada e no campo de dados é enviada a chave criptografada utilizando a chave e o algoritmo anterior. Isso dificulta a captura da chave por partes não autorizadas, no entanto, para trabalhos futuros devem ser considerados outros métodos para permitir uma maior segurança.

Logo após o envio do pacote indicando troca de chave, o servidor realiza a troca do conjunto utilizado no envio dos dados. Quando o cliente recebe a requisição de troca de chave, o conjunto utilizado para receber os dados é alterado pelo novo; o cliente então envia um pacote com o bit 2 do byte de controle (relacionado ao “ack” da chave – que indica que o cliente recebeu e substituiu o conjunto) igual a ‘1’.

Importante destacar que se, durante esta troca, algum pacote for perdido pode haver problemas durante a definição da chave causando falha na comunicação deste ponto em diante. No entanto, o uso do TCP permite uma certa garantia de que os pacotes chegarão ao destino final, minimizando assim a preocupação com este problema.

Tabela 20 – Algoritmos disponíveis no sistema

Valor	Algoritmo
000	AES
001	DES
010	RC6
011	SERPENT
100	TWOFISH
101	MARS
110	AES – Utilizando JNI

Após o envio deste dado de controle o cliente troca também seu conjunto utilizado para enviar os dados finalizando assim a troca no lado cliente. Quando o servidor recebe o pacote de confirmação (“ack” da chave) ele também troca o conjunto utilizado na recepção dos dados.

5.4 – Análise de Desempenho

Foram realizados testes para verificar o impacto que a troca dinâmica de chaves e algoritmos causa para o sistema.

Para os testes de desempenho, foram utilizados três ambientes: **no primeiro** foi utilizado um computador Athlon XP 2400 (2.0 Ghz) com 256 Mb de memória RAM (*Random*

Access Memory) com o SICO executado localmente; **no segundo** ambiente foi utilizado um computador Pentium 4 – 2.8 Ghz e 512 Mb de memória RAM com o SICO executado localmente; **o terceiro ambiente** foi formado por dois computadores Pentium 4 – 2.8 Ghz e 512 Mb de memória RAM com um computador executando a parte cliente e outro a parte servidor do SICO. Para simplificar, estes ambientes são descritos como: Athlon Local, P4-Local e P4-Remoto, respectivamente.

Com isto foi definido como conjunto de testes o uso do SICO para a transmissão de um arquivo de vídeo com o tamanho de 108 Mbytes (113.839.120 bytes) utilizando os algoritmos descritos na seção 5.2.

A Figura 19 mostra uma comparação entre os algoritmos implementados relacionando seu desempenho em cada um dos ambientes de teste definidos. Pode-se perceber que o P4-Local obteve quase sempre o melhor desempenho, entretanto, não foi superior a 1,2 Mbytes/s. Esta velocidade é muito inferior quando comparada aos testes realizados e apresentados no Capítulo 2. Esta lentidão pode estar relacionada ao tempo de comunicação existente no SICO, desta forma, foi realizado um estudo para verificar possíveis otimizações e foi encontrado um possível fator para tal lentidão.

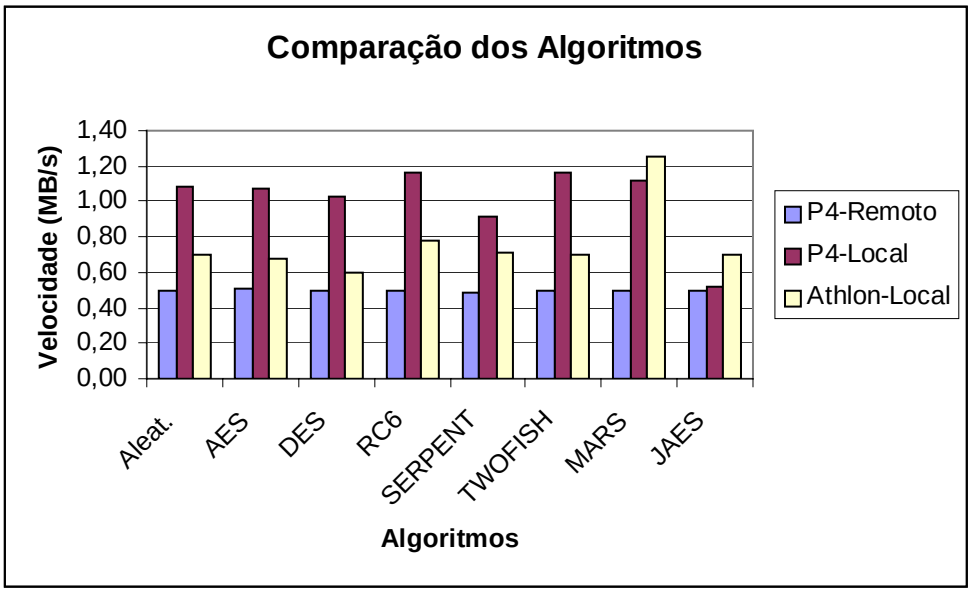

Figura 19 – Comparação dos Algoritmos no SICO

Este fator é o tamanho de um buffer de recepção utilizado pelo sistema. Com isso, foram realizados testes variando o tamanho do buffer entre 64 Kbytes e 8 Mbytes para verificar qual seria o tamanho que melhor atendesse as necessidades do SICO. Estes testes serão apresentados na seção 5.4.1, já na seção 5.4.2 são apresentados testes mais detalhados utilizando o tamanho de buffer mais adequado.

5.4.1. Variação do tamanho do buffer

Para resolver o problema da lentidão do sistema, foram realizados testes variando o tamanho do buffer até que se chegasse ao valor adequado. Nestes testes não foram consideradas as políticas de troca de chave e algoritmo e sim definido um intervalo de tempo em que as trocas devem ocorrer. Foram então definidos testes variando algoritmos e chaves aleatoriamente dentro deste intervalo de tempo.

As Figuras 20, 21 e 22 apresentam a relação entre o tamanho do buffer e o desempenho do sistema verificado através da velocidade média de transmissão. Pode-se perceber que à medida que o tamanho do buffer é incrementado a velocidade média do SICO aumenta. Entretanto, a partir de 1024 Kbytes o desempenho parece estabilizar; e em alguns casos até diminui. Importante destacar que no caso da Figura 22 a tendência é um pouco diferente do que ocorre com os testes locais não demonstrando diminuição do desempenho a medida que o tamanho do buffer aumenta. Com isso, definiu-se como padrão para o SICO o tamanho de buffer de 1024 Kbytes uma vez que este valor permite obter um bom desempenho em ambos os casos.

Essa diferença de tendência ocorre pois quando o buffer possui pouca capacidade de armazenamento, ao ser preenchido, faz com que a comunicação seja suspensa até um determinado instante. Quando a comunicação é retomada existe um acumulo do tempo de latência da rede (tempo de chegada do primeiro pacote). Este tempo de latência é critico para casos de transferência em rede, mas em testes locais a latência é insignificante. Com isto, o tamanho do buffer tem um impacto significativo nos testes envolvendo a rede, sendo que um buffer de maior capacidade resulta em uma menor quantidade de interrupções na comunicação e conseqüentemente um melhor desempenho.

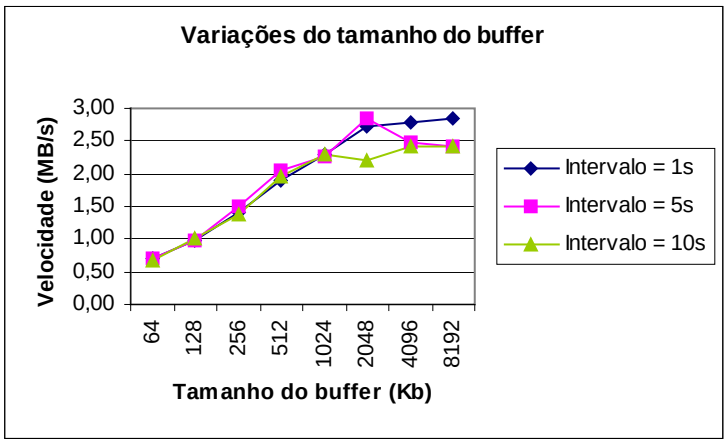

Figura 20 – Variações do tamanho do buffer – Athlon-Local

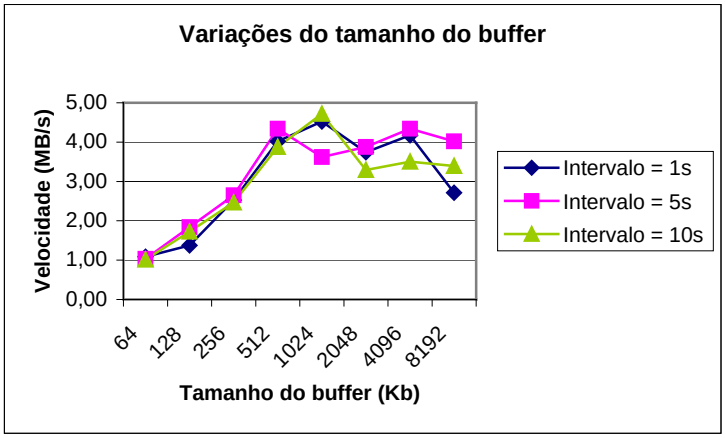

Figura 21 – Variações do tamanho do buffer – P4-Local

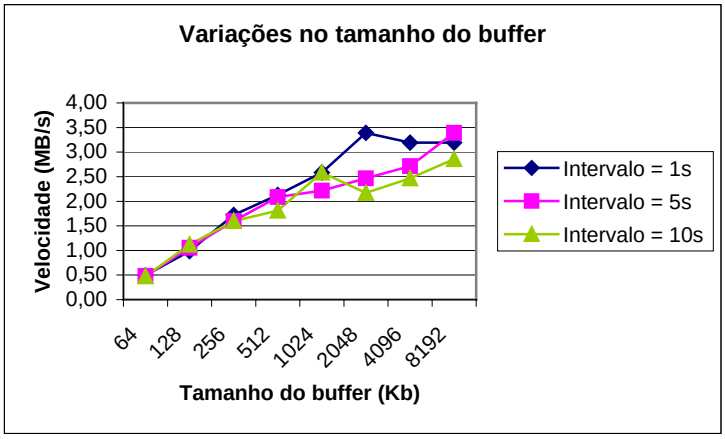

Figura 22 – Variações do tamanho do buffer – P4-Remoto

5.4.2. Testes com o buffer de 1024 Kbytes

Nestes testes ainda não foram consideradas as políticas de troca de chave e algoritmo e sim definido um intervalo de tempo em que as trocas devem ocorrer. Foram então definidos testes com algoritmos fixos variando somente a chave e testes variando chaves e algoritmos aleatoriamente.

As Figuras 23, 24 e 25 apresentam os gráficos que relacionam o intervalo de troca de chaves com o desempenho geral do SICO. Pode-se perceber que à medida que aumenta o intervalo entre as trocas, o número de trocas de chaves diminui, no entanto, o desempenho do sistema não é muito afetado. Isto significa que, ao contrário do que se imagina, neste contexto é viável executar quantas trocas de chave forem necessárias. Desta forma, pode-se concluir que é possível utilizar a troca de chaves e algoritmos dinamicamente sem sofrer grandes perdas de desempenho.

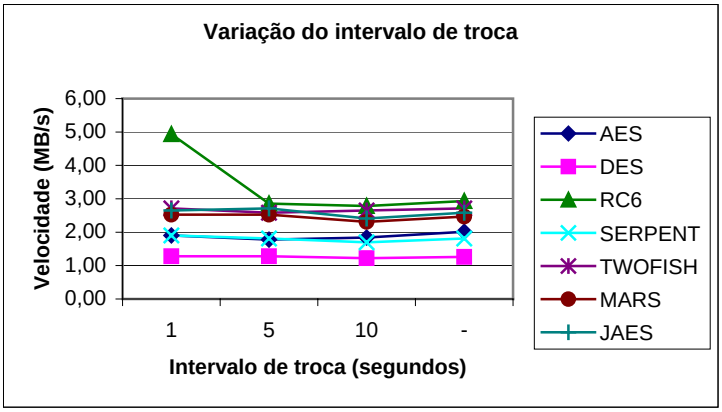

Figura 23 – Variação do intervalo de troca – Athlon-Local

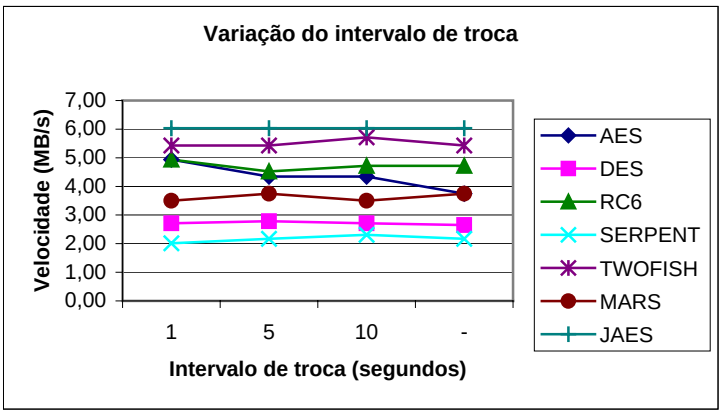

Figura 24 – Variação do intervalo de troca – P4-Local

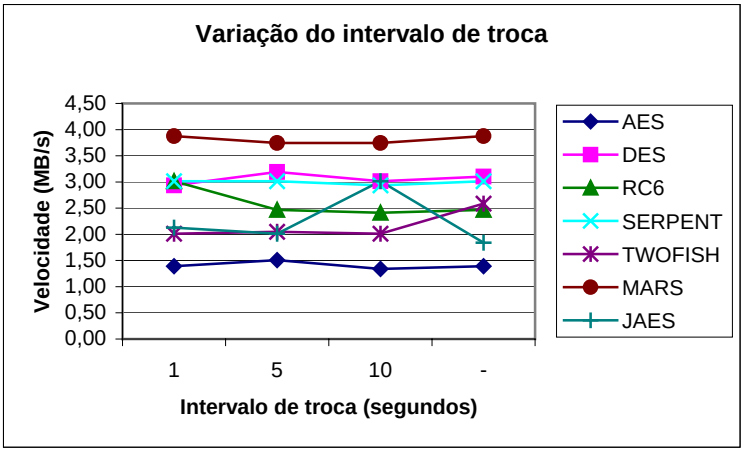

Figura 25 – Variações do intervalo de troca – P4-Remoto

Outro detalhe importante a se perceber observando as Figuras 23, 24 e 25 é que o algoritmo com melhor performance varia para cada caso. Isto mostra a importância da escolha correta do algoritmo.

Desta forma, foi proposta e implementada uma maneira de se escolher o melhor algoritmo no início de uma conexão. Esta escolha foi realizada através de uma análise dos desempenhos dos algoritmos em tempo de execução. Os resultados desta política são apresentados na seção 5.4.3.

5.4.3. Testes realizados utilizando políticas para a troca dos algoritmos

Com base nos resultados anteriores, foi possível identificar que não há um impacto significativo quanto ao número de trocas de chaves e algoritmos. Desta forma foi proposto realizar testes em tempo de execução com todos os algoritmos implementados e verificar qual obteve o melhor desempenho. Após definir o algoritmo com melhor desempenho, o sistema o utiliza, variando somente as chaves utilizadas.

Para verificar qual o algoritmo possui o melhor desempenho no contexto em que está sendo executado, cada algoritmo é utilizado durante o intervalo de 1,5 segundos e é verificado qual algoritmo processou a maior quantidade de dados. Considerando que existem sete algoritmos implementados, o tempo total para testá-los é de 10,5 segundos. Como o teste é executado ao mesmo tempo em que o sistema já está ativo e processando as informações, não é necessário um tempo extra para esta escolha.

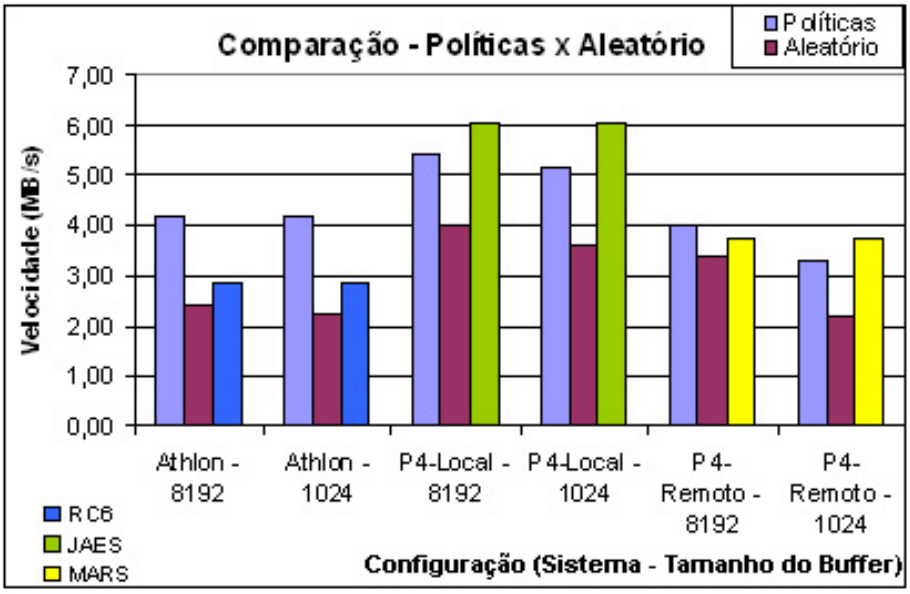

Figura 26 – Testes realizados com uma política para troca de algoritmos

A Figura 26 mostra o gráfico comparando o desempenho obtido com essa política e o obtido trocando os algoritmos e chaves aleatoriamente. Pode-se perceber que esta política obteve sempre o melhor resultado. Entretanto, quando comparamos essa política com os resultados obtidos utilizando somente o RC6, o JAES e o MARS, que foram os mais rápidos para estes ambientes, pode-se perceber que nem sempre a política obteve um melhor resultado.

Mas ainda assim, para alguns casos a política ainda obteve um melhor resultado. Isto pode ocorrer pois o ambiente de execução é dinâmico e o algoritmo criptográfico que foi definido como o mais rápido em um teste da política pode não ser o melhor algoritmo em outros instantes, ou seja, o algoritmo mais rápido no início pode não se manter sempre como o mais rápido. Motivo pelo qual seria interessante fazer mais testes da política de escolha do algoritmo durante uma conexão.

A Figura 27 apresenta a relação entre o tempo total de execução do SICO para este conjunto de testes e o tempo gasto para a escolha do melhor algoritmo a ser utilizado. Pode-se

perceber que em alguns casos, o tempo para a escolha chega a ser 50%, no entanto, quando se compara com os resultados obtidos na Figura 26 verifica-se que ainda assim há uma vantagem na utilização desta política.

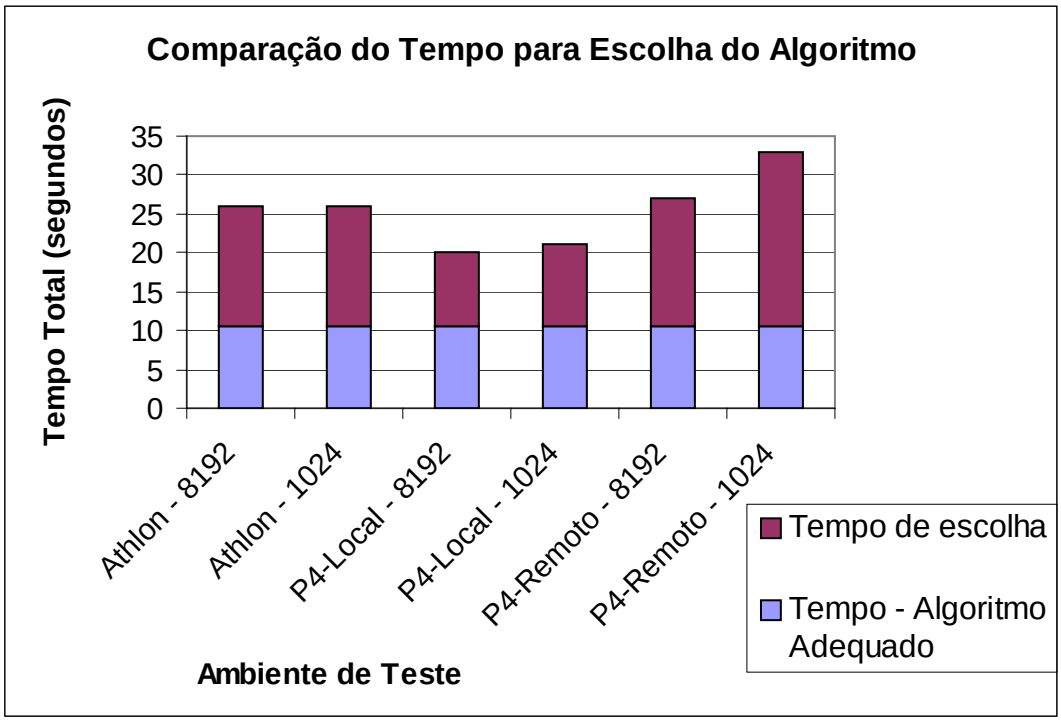

Figura 27 – Relação entre o tempo de escolha e o tempo total

É importante destacar que para esta política não foram considerados aspectos relacionados com a segurança necessária. Para trabalhos futuros, pretende-se incluir métodos que verifiquem junto ao usuário o nível de segurança desejado, e que essa informação reflita para o algoritmo final escolhido.

5.5 – API do SICO

O SICO foi implementado de forma a permitir que outras aplicações sejam criadas utilizando seus recursos. Para isto foi definida uma API em Java para prover acesso aos principais métodos necessários para utilização do SICO.

Como o SICO foi criado utilizando a abordagem cliente-servidor, o lado servidor possui o método responsável por esperar por novas conexões e o lado cliente o método responsável por realizar a conexão com o servidor. Os métodos comuns aos dois lados são os responsáveis pelo envio e recepção dos dados.

Desta forma, quando um objeto do tipo SICO é criado e a conexão estabelecida, é necessário apenas utilizar os métodos para envio e recepção dos dados sem se preocupar com detalhes de rede e segurança, uma vez que esses detalhes são de responsabilidade do SICO.

A descrição dos métodos da API SICO está apresentada no Apêndice A e a API SICO está disponível para *download* na página: www.chiaramonte.pro.br/sico/. Para verificar informações relacionadas sobre como instalar a API e criar aplicações simples utilizando-a, consulte o Apêndice B.

5.6 – Parâmetros Adicionais e Política de Decisão

No primeiro protótipo implementado em linguagem C foram considerados os parâmetros: velocidade de processamento, velocidade da rede e nível de segurança desejado.

Em um estudo mais abrangente foi detectada a existência de vários outros parâmetros que podem influenciar a conexão e sua segurança. Estes parâmetros detectados podem ser classificados em três classes distintas:

- Parâmetros do Sistema: parâmetros como velocidade de rede e velocidade de processamento disponível.
- Parâmetros da Aplicação: parâmetros como o nível de segurança necessário para uma determinada aplicação.
- Parâmetros da Tecnologia: são parâmetros relacionados com a tecnologia física de rede utilizada. Por exemplo, em uma rede *wireless* o nível de segurança é significativamente menor que uma rede ponto a ponto utilizando um cabo de par-trançado, o que demandaria algoritmos mais seguros ou chaves de maior tamanho.

Desta forma, apesar da implementação da política de escolha do algoritmo mais rápido ter apresentado bons resultados, seria interessante, em trabalhos futuros, analisar o impacto da consideração dos parâmetros descritos nesta seção para a escolha do algoritmo a ser utilizado durante a conexão.

CAPÍTULO 6 – CONCLUSÕES E TRABALHOS FUTUROS

Foi apresentada a proposta de um sistema seguro para a transmissão de dados capaz de escolher através de alguns parâmetros, qual o melhor algoritmo a ser utilizado em um determinado ambiente. Um primeiro protótipo foi implementado em linguagem C para validar o modelo. Este protótipo mostrou um bom desempenho utilizando-se de apenas um algoritmo (algoritmo Posicional).

A partir deste protótipo foi desenvolvido o sistema completo considerando os algoritmos criptográficos: AES, DES, RC6, Serpent, Twofish e Mars. Apesar do protótipo ter sido desenvolvido em Linguagem C, a linguagem escolhida para o desenvolvimento do sistema foi a linguagem Java, uma vez que a linguagem C apresentou algumas limitações na manipulação das *threads* e não possui recursos de orientação a objetos.

Foram então realizados testes de desempenho para verificar se o sistema realmente atendia as necessidades impostas para uma comunicação rápida e segura. Através destes testes foi possível observar que o algoritmo escolhido para a comunicação possui uma grande influência para o desempenho total do sistema.

Desta forma, foi então proposto um método para testar o desempenho dos algoritmos implementados em tempo real, sendo que, após este teste o sistema poderia tomar a decisão de escolha do algoritmo criptográfico que mais se adaptou ao ambiente de execução. A implementação deste método permitiu um aumento de até 50% no desempenho total do sistema, do que quando comparado com uma escolha aleatória.

Futuramente, poderiam ser realizadas implementações que verifiquem os parâmetros citados na seção 5.6, comparando os resultados com os obtidos utilizando a política de testes de desempenho em tempo real.

Seria também interessante a realização futura de testes utilizando aplicações que necessitem de grande quantidade de recursos de rede e processamento, como por exemplo, aplicações de vídeo em tempo real.

Uma vez que os resultados apresentados foram satisfatórios, poderiam ser realizados estudos referentes a uma futura implementação do sistema em hardware utilizando FPGAs. Desta forma, uma arquitetura em hardware poderia ser definida para a execução dos algoritmos de criptografia e verificação dos possíveis parâmetros que influenciam no desempenho do sistema.

Uma vez que não foram considerados algoritmos próprios para distribuição de chaves, bem como algoritmos assimétricos utilizando o esquema de chave pública, seria interessante verificar o impacto gerado no sistema por esses algoritmos, desta forma, poderia ser realizada a implementação de tais algoritmos e a realização de testes detalhados para a verificação do desempenho deste tipo de abordagem.

Outra consideração importante de ser verificada futuramente está relacionada com a implementação de diferentes níveis de segurança e a determinação do algoritmo a ser utilizado com base nesses níveis. Seria interessante também a construção de um sistema que auxilie na determinação do nível de segurança requerido por determinado usuário ou aplicação.

REFERÊNCIAS

AESWINNER. **National Institute of Standards and Technology**, NIST. <www.nist.gov/public_affairs/releases/g00-176.htm>. Acesso em 12 Dez 2003.

BORISOV, N.; GOLDBERG, I.; WAGNER, D., **"Intercepting Mobile Communications: The Insecurity of 802.11**.", in Proc. of 7th International Conference on Mobile Computing and Networking, MOBICOM'01, ACM Press, pp. 180-189, 2001

CHIARAMONTE, R. B., **Implementação e Teste em Hardware e Software de Sistemas Criptográficos**. Trabalho de Conclusão de Curso (Apoio FAPESP nº 00/14166-8). UNIVEM – Centro Universitário Eurípides de Marília – Marília, 2003.

CHIARAMONTE, R.B.; MORENO, E.D., **Criptografia Posicional em Hardware (VHDL e FPGAs)**. Revista REIC-SBC (Revista Eletrônica de Iniciação Científica), Sociedade Brasileira de Computação . Ano II, Vol. II, No. IV, Dez. 2002, ISSN: 1519-8219.

CHIARAMONTE, R.B.; MORENO, E.D**., Sico - Um Sistema Inteligente e Adaptativo Para Comunicação de Dados: Resultados Preliminares**; In: GCETE - Global Congress on Engineering and technology Education, Bertioga, 2005.

CHIARAMONTE, R.B.; PEREIRA, F.D.; MORENO, E.D.; **Um Algoritmo Criptográfico Posicional – Otimizações e Desempenho**; Revista de Engenharia de Computação e Sistemas Digitais; ISSN: 1678-8435; número 2 – Novembro 2005.

DAEMEN, J.; RIJMEN, V. **AES Proposal: Rijndael Block Cipher**. 03/09/99

DIERKS, T.; ALLEN, C., **The TLS Protocol Version 1.0**. RFC 2246 –Janeiro 1999.

DIFFIE, W.; HELLMAN, M., **New Directions in cryptography**. IEEE Trans. Inform. Theory IT-22, (Nov. 1976), 644-654.

FIPS-PUB46-3, "**DATA ENCRYPTION STANDARD (DES)**", Federal Information Processing Standard (FIPS), NIST, 1999

FREIER, A.O.; KARLTON, P.; KOCHER, P.C., **The SSL Protocol Version 3.0**. Internet Draft –Disponível em: http://wp.netscape.com/eng/ssl3/draft302.txt. Acesso: 04/2004

GLADMAN, B. **AES Second Round Implementation Experience**. Disponível em: <http://fp.gladman.plus.com/cryptography_technology/aesr2/index.htm>. Acesso em: 10 fev 2006

GUELFI, A. E., **Middleware de Comunicação para Prover o Gerenciamento de Serviços Multimídia Flexíveis e Integrados em Ambientes Distribuídos e Heterogêneos.** Tese de Doutorado. EPUSP – Escola Politécnica da Universidade de São Paulo – São Paulo, 2002.

INTELLON – **No New Wires**. Site da Web: http://www.intellon.com/. Acesso em Janeiro de 2005.

LAI, X.; MASSEY, J. L., **A proposal for a new block encryption standard**. Aarhus: Springer Verlag, 1991.

MATSUI, M., **Linear Criptanalysis Method for DES Cipher**. In advances in Criptology, Eurocrypt 93, Lectures Notes in Computer Science Vol. 765, Springer Verlag, pp. 386-397, 1993.

MENEZES, A.; OORSCHOT, P.V.; VANSTONE, S. **Handbook of Applied Cryptography**. New York: CRC Press, 1997.

MEYLAN, F., **CriptoQoS: Uma Plataforma de Gerenciamento e Desenvolvimento de Aplicações Distribuídas com Suporte Integrado à Qos e Segurança**. Tese de Doutorado. EPUSP – Escola Politécnica da Universidade de São Paulo – São Paulo, 2003.

MIERS, C. C.; CUSTODIO, R. F.. **Modelo Simplificado do Cifrador AES**. Florianópolis: UFSC, 2002.

MORENO, E.D.; PEREIRA, F.D.; CHIARAMONTE, R.B., **Criptografia em Software e Hardware**. São Paulo, Novatec Editora, 2005.

PEREIRA, F.D.; MORENO, E.D. **Otimização em VHDL e Desempenho em FPGAs do Algoritmo de Criptografia DES**. Quarto Workshop em sistemas computacionais de alto desempenho (WSCAD). São Paulo, 2003.

RFC1321. RFC 1321 – **The MD5 Message Digest Algorithm**, 1992.

RFC3174. RFC 3174 – **US Secure Hash Algorithm 1 (SHA-1).** NSA, 1994.

RIVEST, R.L.; SHAMIR, A.; ADLEMAN, L., **A Method for Obtaining Digital Signatures and Public-Key Cryptosystems**. Communications of the ACM, 21(2):120-126, February 1978.

ROSE. O. "**Statistical Properties of MPEG Video Traffic and their Impact on Traffic Modeling in ATM Systems**". In Proceedings of the 20th Annual Conference on Local Computer Networks, 1995

SANTOS, A.P.S. dos, **"Qualidade de Serviço na Internet"**, Boletim bimestral sobre tecnologia de redes – RNP (Rede Nacional de Ensino e Pesquisa), Volume 3, Número 6, ISSN 1518-5974, 12 de novembro de 1999. Disponível em: http://www.rnp.br/newsgen/9911/qos.html

SCHNEIER, B. **Applied Cryptography: Protocols, Algorithms and Source in C.** 2nd ed. New York: John Wiley and Sons, 1996.

STALLINGS, W. **Cryptography and Network Security - Principles and Practice**, Second Edition. Prentice Hall, 1998

SUN - **Glossary of Terms.** Site da Web. Consulta realizada em Dezembro 2005: http://java.sun.com/docs/books/tutorial/information/glossary.html

TANENBAUM, A. S. “**Computer Networks**”, Third Edition, Prentice Hall, New Jersey, 1996;

TANENBAUM, Andrew S.; WOODHULL, Albert S.. **Sistemas operacionais : projeto e implementação.** 2ª ed. Porto Alegre: Bookman, 2000. 759p.

TERADA, R. **Segurança de Dados – Criptografia em Redes de Computadores**, 1ª ed. Edgard Blücher, 2000.

TKOTZ, V. **Criptografia Numaboa**. Disponível em: <http://www.numaboa.com.br/criptologia>. Acesso em 12 Dez 2003.

WALRAND, J.; VARAIYA, P.; “**High-Performance Communication Networks**”, Second Edition, Academic Press, San Diego, 2000;

XIAO, X., e NI, M. N., "**Internet QoS: A Big Picture**", IEEE Network, March/April 1999

APÊNDICES

Apêndice A – API do Sistema Inteligente de Comunicação.

SICO
Class SICOs
`java.lang.Object`
  └ **`SICO.SICOs`**

---

public class **SICOs**
extends java.lang.Object

Esta classe implementa o servidor do Sistema Inteligente de comunicação.

Constructor Summary
`SICOs``(int porta)` Cria um objeto do tipo SICO Server especificando uma porta para as conexões.
`SICOs``(int porta, int backlog)` Cria um objeto do tipo SICO Server especificando uma porta para as conexões e indicando o tamanho máximo da fila de conexões.
`SICOs``(int porta, int backlog, java.net.InetAddress localAddr)` Cria um objeto do tipo SICO Server especificando uma porta e um endereço para as conexões e indicando o tamanho máximo da fila de conexões.
`SICOs``(int porta, int backlog, java.net.InetAddress localAddr, int nivelMinimo, int nivelMaximo)` Cria um objeto do tipo SICO Server especificando uma porta e um endereço para as conexões, indicando o tamanho máximo da fila de conexões e níveis de segurança admitidos.

Method Summary	
`void`	**`close`**`()` Termina uma conexão finalizando todos os médotos de entrada e saída utilizados pelo SICO.
`void`	**`envia`**`(byte[] b, int nbytes)` Envia dados através da conexão segura estabelecida pelo SICO.

void	**enviaFinal**() Finaliza o envio dos dados.
void	**esperaConexao**() Aguarda por uma conexão, quando uma conexão é estabelecida com sucessom o método termina a execução retornando a seqüência de execução para a linha seguinte à sua chamada.
int	**recebe**(byte[] b) Recebe dados através da conexão segura estabelecida pelo SICO.
int	**recebeFinal**(byte[] b) Finaliza o recebimento dos dados.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

SICOs

```
public SICOs(int porta)
      throws java.io.IOException,
             java.lang.SecurityException
```

Cria um objeto do tipo SICO Server especificando uma porta para as conexões.

Parameters:

`porta` - número da porta para conexões.

SICOs

```
public SICOs(int porta,
             int backlog)
      throws java.io.IOException,
             java.lang.SecurityException
```

Cria um objeto do tipo SICO Server especificando uma porta para as conexões e indicando o tamanho máximo da fila de conexões.

Parameters:

`porta` - número da porta para conexões.

`backlog` - tamanho da fila de conexões.

SICOs

```
public SICOs(int porta,
             int backlog,
             java.net.InetAddress localAddr)
      throws java.io.IOException,
             java.lang.SecurityException
```

Cria um objeto do tipo SICO Server especificando uma porta e um endereço para as conexões e indicando o tamanho máximo da fila de conexões.

Parameters:

`porta` - número da porta para conexões.

`backlog` - tamanho da fila de conexões.
`localAddr` - endereço local.

SICOs

```
public SICOs(int porta,
             int backlog,
             java.net.InetAddress localAddr,
             int nivelMinimo,
             int nivelMaximo)
      throws java.io.IOException,
             java.lang.SecurityException
```

Cria um objeto do tipo SICO Server especificando uma porta e um endereço para as conexões, indicando o tamanho máximo da fila de conexões e níveis de segurança admitidos.

Parameters:

`porta` - número da porta para conexões.
`backlog` - tamanho da fila de conexões.
`localAddr` - endereço local.
`nivelMinimo` - nível mínimo de segurança.
`nivelMaximo` - nível máximo de segurança.

Method Detail

esperaConexao

```
public void esperaConexao()
                  throws java.io.IOException
```

Aguarda por uma conexão, quando uma conexão é estabelecida com sucessom o método termina a execução retornando a seqüência de execução para a linha seguinte à sua chamada. A partir deste momento podem ser utilizados os métodos para o envio e recebimento de dados.

Throws:

`java.io.IOException`

---

close

```
public void close()
           throws java.io.IOException
```

Termina uma conexão finalizando todos os médotos de entrada e saída utilizados pelo SICO. Antes de finalizar uma conexão é necessário finalizar todas as transmissões de dados pendentes através de chamadas aos métodos enviaFinal() e recebeFinal()

Throws:

`java.io.IOException`

See Also:

`enviaFinal()`, `recebeFinal(byte [] b)`

---

enviaFinal

```
public void enviaFinal()
                throws java.io.IOException
```

Finaliza o envio dos dados. Este método deve ser chamado antes de finalizar a conexão quando houver dados a enviar.

Throws:

`java.io.IOException`

See Also:
close()

---

envia

```
public void envia(byte[] b,
                  int nbytes)
           throws java.io.IOException
```

Envia dados através da conexão segura estabelecida pelo SICO.

Parameters:

`b` - vetor de bytes para serem enviados.

`nbytes` - número de bytes a serem enviados.

Throws:

`java.io.IOException`

---

recebeFinal

```
public int recebeFinal(byte[] b)
                throws java.io.IOException
```

Finaliza o recebimento dos dados. Este método deve ser chamado antes de finalizar a conexão quando houver dados a receber.

Parameters:

`b` - vetor onde serão armazenados os dados recebidos

Returns:

o número de bytes que foram recebidos

Throws:

`java.io.IOException`

See Also:

close()

---

recebe

```
public int recebe(byte[] b)
           throws java.io.IOException
```

Recebe dados através da conexão segura estabelecida pelo SICO.

Parameters:

`b` - vetor onde serão armazenados os dados recebidos

Returns:

o número de bytes que foram recebidos

Throws:

`java.io.IOException`

SICO
Class SICOc
```
java.lang.Object
  └SICO.SICOc
```

---

public class **SICOc**
extends java.lang.Object

Esta classe implementa o cliente do Sistema Inteligente de comunicação.

Constructor Summary

SICOc(java.lang.String servidor, int porta, int nivel) Cria um objeto do tipo SICO Client atribuindo os dados necessários para uma conexão com um servidor SICO.

Method Summary

void	**close**() Termina uma conexão finalizando todos os médotos de entrada e saída utilizados pelo SICO.
void	**conecta**() Realiza a conexão a partir dos dados passados ao construtor.
void	**envia**(byte[] b, int nbytes) Envia dados através da conexão segura estabelecida pelo SICO.
void	**enviaFinal**() Finaliza o envio dos dados.
int	**recebe**(byte[] b) Recebe dados através da conexão segura estabelecida pelo SICO.
int	**recebeFinal**(byte[] b) Finaliza o recebimento dos dados.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

SICOc

```
public SICOc(java.lang.String servidor,
             int porta,
```

```
        int nivel)
    throws java.net.UnknownHostException
```

Cria um objeto do tipo SICO Client atribuindo os dados necessários para uma conexão com um servidor SICO.

Parameters:

`servidor` - endereço do servidor para a conexão.

`porta` - número da porta a se conectar no servidor.

`nivel` - nível de segurança desejado para a conexão.

Method Detail

conecta

```
public void conecta()
            throws java.io.IOException,
                   java.lang.SecurityException
```

Realiza a conexão a partir dos dados passados ao construtor. Quando uma conexão é estabelecida com sucesso o método termina a execução retornando a seqüência de execução para a linha seguinte à sua chamada. A partir deste momento podem ser utilizados os métodos para o envio e recebimento de dados.

Throws:

`java.io.IOException`

`java.lang.SecurityException`

---

close

```
public void close()
          throws java.io.IOException
```

Termina uma conexão finalizando todos os médotos de entrada e saída utilizados pelo SICO. Antes de finalizar uma conexão é necessário finalizar todas as transmissões de dados pendentes através de chamadas aos métodos enviaFinal() e recebeFinal()

Throws:

`java.io.IOException`

See Also:

`enviaFinal()`, `recebeFinal(byte [] b)`

---

enviaFinal

```
public void enviaFinal()
                throws java.io.IOException
```

Finaliza o envio dos dados. Este método deve ser chamado antes de finalizar a conexão quando houver dados a enviar.

Throws:

`java.io.IOException`

See Also:

`close()`

---

envia

```
public void envia(byte[] b,
                  int nbytes)
           throws java.io.IOException
```

Envia dados através da conexão segura estabelecida pelo SICO.

Parameters:

`b` - vetor de bytes para serem enviados.

`nbytes` - número de bytes a serem enviados.

Throws:

java.io.IOException

---

recebeFinal

```
public int recebeFinal(byte[] b)
                throws java.io.IOException
```

Finaliza o recebimento dos dados. Este método deve ser chamado antes de finalizar a conexão quando houver dados a receber.

Parameters:

b - vetor onde serão armazenados os dados recebidos

Returns:

o número de bytes que foram recebidos

Throws:

java.io.IOException

See Also:

close()

---

recebe

```
public int recebe(byte[] b)
           throws java.io.IOException
```

Recebe dados através da conexão segura estabelecida pelo SICO.

Parameters:

b - vetor onde serão armazenados os dados recebidos

Returns:

o número de bytes que foram recebidos

Throws:

java.io.IOException

Apêndice B – Criando uma aplicação utilizando o SICO

Para criar uma aplicação utilizando o SICO é necessário criar um servidor e um cliente. Inicialmente será discutido como criar a parte do servidor utilizando o SICO, para isso, será utilizada como base a API do sistema descrita no Apêndice A.

A primeira etapa é a criação de um objeto do tipo SICOs, para isso, neste exemplo, será utilizado o construtor onde se informa somente o número da porta para realizar as conexões. Desta forma, a instanciação do objeto servidor é:

```
SICOs serv = new SICOs(12345);
```

Após instanciar o objeto é necessário esperar até que uma conexão seja estabelecida. Para isso, basta invocar o método esperaConexão() presente na classe SICOs como segue:

```
serv.esperaConexão();
```

Feito isso basta utilizar os métodos responsáveis pelo envio e recebimento dos dados para realizar a comunicação. Importante destacar que a parte de criptografia é realizada internamente no sistema, sendo assim transparente ao usuário.

A Figura 28 apresenta o código completo da implementação de um servidor utilizando o SICO.

```
import java.io.*;
public class exemploServ
{
	public static void main( String args[] )
	{
		try
		{
			// criando o servidor e esperando a conexão
			SICOs serv = new SICOs( 12345 );
			serv.esperaConexao();
			// recebendo dados
			byte b[] = new byte[20];
			int recebeu = serv.recebe(b);
			System.out.println("recebeu: "+ recebeu + " bytes");
			System.out.println("dados: " + new String(b, 0, recebeu));
			System.out.println("--- recebendo final ---");
			recebeu = serv.recebeFinal(b);
			System.out.println("recebeu: "+ recebeu + " bytes");
			System.out.println("dados: " + new String(b, 0, recebeu));
			// fechando a conexão
			serv.close();
		}
		catch(IOException e)
		{
			System.err.println("erro de IO");
		}
	}
}
```

Figura 28 – Código completo de um servidor utilizando o SICO

A segunda etapa consiste em criar um cliente para realizar a conexão ao servidor do SICO. Da mesma forma que o servidor, também é necessário instanciar um objeto do tipo SICOc (cliente SICO) e chamar o método responsável por realizar a conexão, neste caso, o método conecta(). Um exemplo de como isto pode ser realizado é:

```
SICOc client = new SICOc(“127.0.0.1", 12345, 3);
client.conecta();
```

Após a realização destes passos é possível realizar a comunicação utilizando os métodos para envio e recebimento de dados, como descrito na API apresentada no Apêndice A.

O código completo de um exemplo de cliente utilizando o SICO é apresentado na Figura 29.

```
import java.io.*;
public class exemploClient
{
	public static void main( String args[] )
	{
		try
		{
			// criando o cliente e realizando a conexão
			SICOc client = new SICOc( "127.0.0.1", 12345, 3 );
			client.conecta();
			// enviando dados
			byte b[] = {‘t’, ‘e’, ‘s’, ‘t’, ‘e’};
			client.envia(b,5);
			client.enviaFinal();
			// fechando a conexão
			client.close();
		}
		catch(IOException e)
		{
			System.err.println("erro de IO");
		}
	}
}
```

Figura 29 – Código completo de um cliente utilizando o SICO

Importante lembrar que em ambos os lados (cliente e servidor), ao final da transmissão é necessário realizar uma chamada ao método close() para finalizar a conexão.