FUNDAÇÃO DE ENSINO “EURÍPIDES SOARES DA ROCHA”
CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA - UNIVEM
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

VASCO MARTINS CORREIA

SERIALIZAÇÃO DOS DADOS DE IMAGENS MÉDICAS USANDO TROCA DE MENSAGENS

MARÍLIA
2005

VASCO MARTINS CORREIA

SERIALIZAÇÃO DOS DADOS DE IMAGENS MÉDICAS USANDO TROCA DE MENSAGENS

Monografia apresentada ao Curso de Ciência da Computação da Fundação de Ensino Eurípides Soares da Rocha, Mantenedora do Centro Universitário Eurípides de Marília - UNIVEM, para obtenção do grau de Bacharel em Ciência da Computação.

Orientador:
Prof. Dr. Marcos Luiz Mucheroni

MARÍLIA
2005

CORREIA, Vasco Martins Correia.
Serialização dos Dados de Imagens Médicas usando Troca de Mensagens / Vasco Martins Correia; orientador: Marcos Luiz Mucheroni. Marilia, SP: [s.n.], 2005.
60 f.

Monografia (Bacharelado em Ciência da Computação) — Centro Universitário Eurípides de Marília – Fundação de Ensino Eurípides Soares da Rocha.

1. Programação Distribuída 2. Serialização de Objetos 3. Imagens Médicas
CDD: 004.6

AGRADECIMENTOS

Agradeço primeiramente Deus, pelo dom da vida.

Aos meus pais, que apesar das dificuldades nunca desanimaram e sempre batalharam para que eu estudasse.

A Alexandra, o grande amor da minha vida. Agradeço a ela por estar sempre ao meu lado, me incentivando e me ajudando, pela sua dedicação e amor. A ela eu dedico todas as minhas conquistas.

Agradeço ao Professore Marcos Luiz Mucheroni, não só pela orientação, mas também pela amizade e paciência.

Agradeço a todos os meus amigos da faculdade que de uma forma ou outra me ajudaram e sempre estiveram ao meu lado.

Agradeço a professora Kalinka Regina Lucas Jaquie Castelo Branco, pela atenção e ajuda, que contribuíram para engrandecer a qualidade deste projeto de pesquisa. Agradeço também a todos os professores que contribuíram para a minha formação acadêmica.

Agradeço em especial a Fundação de Amparo a Pesquisa do Estado de São Paulo (FAPESP), pelo financiamento desse projeto de pesquisa (Processo Nº 03/07557-9).

CORREIA, Vasco Martins. **Serialização dos Dados de Imagens Médicas usando Troca de Mensagens**. 2005. 60 f. Monografia (Bacharelado em Ciência da Computação) - Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2005.

RESUMO

O presente trabalho tem como objetivo desenvolver um ambiente distribuído para busca e armazenamento de imagens médicas, que tem como principal finalidade a comunicação de máquinas clientes com um servidor, utilizando para o envio dos dados o Mecanismo de Serialização de Objetos. Foram realizados testes a fim de medir o desempenho da aplicação proposta com uma aplicação desenvolvida que transfere a imagem médica sem o uso da Serialização de Objetos. A análise estatística utilizada para verificar se as diferenças de desempenho das aplicações são estatisticamente significativas foi a denominada Teste de Hipóteses. Mediante a pesquisa desenvolvida conclui-se que o uso da Serialização de Objetos permite ganhos reais de desempenho na camada de transporte estatisticamente significativos quando comparados com aplicações que não utilizem a Serialização de Objetos, como pode ser observado pelos valores do Teste de Hipótese.

Palavras-chaves: Programação Distribuída. Serialização de Objetos. Imagens Médicas.

LISTA DE ILUSTRAÇÕES

Figura 1 – Exemplo de implementação com o mpiJAVA 13
Figura 2 – Sintaxe do método writeObject() 14
Figura 3 – Sintaxe do método readObject() 14
Figura 4 – Exemplo de classe serializavel 15
Figura 5 – Cliente Serialização 16
Figura 6 – Exemplo de Servidor de Serialização 17
Figura 7 – Tela de login do sistema 23
Figura 8 – Tela de cadastro de novo usuário 23
Figura 9 – Comando para carregar o Driver 24
Figura 10 – Comando para realizar a conexão com o Banco de Dados 24
Figura 11 – Objeto da classe Statement 25
Figura 12 – Parte do código de login do usuário ao sistema 26
Figura 13 – Interface do Cliente 27
Figura 14 – Visualização da Imagem 29
Figura 15 – Gráfico da média dos tempos de transporte 30
Figura 16 – Formula estatística do Teste de Hipóteses 32
Figura 17 – Média da serialização em comparação com a não serialização (amostra 1) 33
Figura 18 – Média da serialização em comparação com a não serialização (amostra 2) 33
Figura 19 – Média da serialização em comparação com a não serialização (amostra 3) 34
Figura 20 – Média da serialização em comparação com a não serialização (amostra 4) 34
Figura 21 – Média da serialização em comparação com a não serialização (amostra 5) 35
Figura 22 – Média da serialização em comparação com a não serialização (amostra 6) 35

Figura 23 – Média da serialização em comparação com a não serialização (amostra 7) 36

Figura 24 – Média da serialização em comparação com a não serialização (amostra 8) 36

Figura 25 – Média da serialização em comparação com a não serialização (amostra 9) 37

LISTA DE TABELAS

Tabela 1 – Valores calculados para o desempenho obtido com o transporte Serializado e o Não Serializado (amostra 1) 33

Tabela 2 – Valores calculados para o desempenho obtido com o transporte Serializado e o Não Serializado (amostra 2) 34

Tabela 3 – Valores calculados para o desempenho obtido com o transporte Serializado e o Não Serializado (amostra 3) 34

Tabela 4 – Valores calculados para o desempenho obtido com o transporte Serializado e o Não Serializado (amostra 4) 35

Tabela 5 – Valores calculados para o desempenho obtido com o transporte Serializado e o Não Serializado (amostra 5) 35

Tabela 6 – Valores calculados para o desempenho obtido com o transporte Serializado e o Não Serializado (amostra 6) 35

Tabela 7 – Valores calculados para o desempenho obtido com o transporte Serializado e o Não Serializado (amostra 7) 36

Tabela 8 – Valores calculados para o desempenho obtido com o transporte Serializado e o Não Serializado (amostra 8) 36

Tabela 9 – Valores calculados para o desempenho obtido com o transporte Serializado e o Não Serializado (amostra 9) 37

LISTA DE ABREVIATURAS E SIGLAS

DARPA - *Defense Advanced Research Projects Administration*

GIS – *Geographical Information Systems*

IP – *Internet Protocol*

JMPI – *Java Message Passing Interface*

JNI – *Java Native Interface*

JVM – *Java Virtual Machine*

MPI – *Message Passing Interface*

PC – *Personal Computer*

PVM – *Parallel Virtual Machine*

RMI – *Remote Method Invocation*

TCP – *Transmission Control Protocol*

SUMÁRIO

CAPÍTULO 1 – INTRODUÇÃO 1

CAPÍTULO 2 – AMBIENTES DISTRIBUÍDOS, MESSAGE PASSING INTERFACE (MPI) E SERIALIZAÇÃO 5

2.1 O Ambiente de Troca de Mensagens MPI 6
2.1.1 Funções Básicas do MPI 7
2.1.2 Compilação e Execução 10

2.2 Ambientes Distribuídos em Java (JMPI e mpiJava) 10

2.3 Recursos de Serialização em Java 13

CAPÍTULO 3 – IMPLEMENTAÇÃO 19

3.1 Ambiente de Serialização de Objetos 19
3.1.1 Classe Serializavel 19
3.1.2 Módulo Servidor 20
3.1.3 Módulo Cliente 22
3.1.3.1 Banco de Dados 23
3.1.3.1.1 Carregamento do Driver 24
3.1.3.1.2 Realização da Conexão 24
3.1.3.1.3 Consultas ao Banco de Dados 25
3.1.3.2 Interface do Cliente 26
3.1.3.3 Visualizando a Imagem 28

3.2 Ambiente MPI 29

CAPÍTULO 4 - RESULTADOS 30

4.1 Teste de Hipótese 31

CAPÍTULO 5 – CONCLUSÕES 38

REFERÊNCIAS 40

APÊNDICE 42

APÊNDICE A – Classe Server 42

APÊNDICE B – Classe Client 44

APÊNDICE C – Classe SerializeImage 48
APÊNDICE D – Classe Login 49
APÊNDICE E – Classe ConnectionDatabase 52
APÊNDICE F – Classe Register 55
APÊNDICE G – Classe CreateImage..... 59
APÊNDICE H – Classe MyCanvas..... 60

CAPÍTULO 1 – INTRODUÇÃO

A Internet foi criada a partir da necessidade de computadores distantes fisicamente pudessem transferir dados e informações entre si de forma rápida, eficiente e segura.

Projetada inicialmente como uma rede privada pelo DARPA (*Defense Advanced Research Projects Administration*) sendo chamada de Darpanet. Conseqüentemente vendo a grande possibilidade de troca de informações e serviços que começavam a surgir, iniciou-se a difundir esta rede também para universidades, centros de pesquisas e entre outros, passando a ser chamada de Arpanet.

A grande quantidade de redes que começaram a surgir e a se interconectar, vieram a se tornar hoje o que chamamos de Internet, ou seja, um conjunto de redes que trocam informações entre si, através de servidores que disponibilizam serviços e de clientes que buscam por informações.

Uma das influências da Internet consiste no aumento da necessidade de uma maior e melhor interação entre aplicações distribuídas. Os desenvolvedores têm tentado responder à essa demanda com uma variedade de produtos Web que permitam ao usuário expandir seu alcance aos recursos da Rede. Dentro desse contexto, para que aplicações servidoras distribuídas ofereçam uma solução durável e de grande alcance, é preciso que possuam duas habilidades importantes: interoperabilidade e portabilidade. Com suporte adequado, tais aplicações devem, então, ser capazes de interagir com outras, independente do equipamento ou sistema operacional em que se encontrem, bem como mudar de plataforma base de forma transparente sempre que necessário.

Os modelos e algoritmos mais utilizados para a criação de aplicações distribuídas são encontrados no paradigma que se denominou, inicialmente, na história da computação, de computação paralela e concorrente. Os sistemas distribuídos são herdeiros destes sistemas

compostos por vários processadores que operam em paralelo ou concorrentemente, e uma vez que os processadores estão interconectados por uma rede, esta computação distribuída também tem uma caracterização paralela (processadores que executam tarefas simultaneamente), realizando assim a cooperação na execução de uma determinada tarefa, sendo o seu objetivo principal, a flexibilidade e desempenho. Já nas arquiteturas paralelas, o objetivo principal é o aumento da capacidade de processamento, utilizando um grande número de processadores, sendo a comunicação realizada por meio de redes especiais de conexão ou através de memória compartilhada. Um exemplo dessas arquiteturas são as estações de trabalho ou PCs conectados em rede, podendo ser exploradas conexões de alto desempenho.

Na origem destes ambientes dois tipos de software se popularizaram para estas máquinas: o ambiente de troca de mensagens MPI (*Message Passing Interface*) (MPI, 1994) e o PVM (*Parallel Virtual Machine*) (BEG, 1994). O PVM disponibiliza um software que pode ser executado em ambientes diferentes, gerando com isso segurança ao criar aplicações paralelas, tendo em vista a portabilidade, originalmente programado em C, mas também são conhecidos ambientes usando Java. Uma tecnologia importante, pela influência nos conceitos e nos algoritmos de computação distribuída, é o MPI (PACHECO, 1997).

MPI, como o próprio nome indica é uma interface de troca de dados, cujo ambiente possibilita uma troca cooperativa entre os processos, porém a forma de tratar os dados é parte essencial deste ambiente. Este padrão é a biblioteca de troca mensagem escolhida para estudo neste trabalho.

Algumas áreas de aplicação encontraram neste novo enfoque um impulso enorme e novas tecnologias têm se agregado a elas, tais como, sistemas GIS (*Geographical Information Systems*), medicina, comércio eletrônico, além de outras.

Uma aplicação que tem encontrado grande desenvolvimento na Web é o tratamento de imagens e serviços relacionados a elas, devido ao tamanho dos arquivos e à necessidades

de desempenho, os estudos podem ser divididos em três grandes áreas de pesquisa: a codificação, o transporte e o processamento digital de imagem.

Tanto PVM quando MPI, já têm ambientes desenvolvidos em Java, chamados de JPVM (FERRARI, 1998) e mpiJava (CARPENTER et all, 1998) (mpiJava, 1999), entre outros desenvolvimentos recentes destes projetos podem destacar-se o MPICH (MPICH, 1996) e o MPJ (CARPENTER et all, 2000).

Os estudos desenvolvidos por (CARPENTER, Fox, Sung & Sang, 1999) mostram a eficiência e segurança no transporte pela rede de comunicação de dados implementados com uso do Mecanismo de Serialização comparados com o transporte com o protocolo TCP/IP, UDP e com o RMI.

1.1 Objetivos

Este trabalho tem por objetivo desenvolver um ambiente que propiciasse o transporte de dados médicos de forma serializada, utilizando troca de mensagens em MPI, através de um ambiente distribuído, que terá como principal finalidade à comunicação de máquinas clientes com um servidor, utilizando para o envio dos dados o Mecanismo de Serialização de Objetos.

1.2 Organização do Texto

O primeiro capítulo faz uma introdução ao trabalho especificando o objetivo do mesmo.

O segundo capítulo aborda o estudo da revisão bibliográfica utilizada neste trabalho.

O terceiro capítulo trata da implementação do ambiente proposto, discutindo cada particularidade do mesmo.

Por fim, no último capítulo trataremos dos resultados obtidos com os testes realizados no ambiente desenvolvido a fim de medir a performance no transporte pela rede.

CAPÍTULO 2 – AMBIENTES DISTRIBUÍDOS, MESSAGE PASSING INTERFACE (MPI) E SERIALIZAÇÃO

Na programação distribuída, usando a arquitetura Cliente/Servidor, clientes e servidores podem ser implementados usando qualquer linguagem de programação que dispõe recursos para a programação distribuída. Nessas aplicações, os programas são executados em diferentes processadores fortemente ou fracamente acoplados, e trocam informações através de uma rede de comunicação. Nesta arquitetura o programa cliente solicita serviços ao programa servidor que, após executá-los, envia respostas ao cliente.

Devido à limitação de sistemas simples de computação, a necessidade de aumentar a velocidade de processamento e do desempenho das máquinas levou-se ao aparecimento dos sistemas de processamento paralelo.

Com a evolução destes sistemas paralelos, baseados em múltiplos processadores, surgiram as redes de computadores e por conseqüência os sistemas distribuídos.

As vantagens dos sistemas distribuídos com uma arquitetura Cliente/Servidor são várias em relação aos sistemas centralizados, dentre elas as mais relevantes são: economia; velocidade; distribuição inerente; crescimento incremental; compartilhamento de dados e dispositivos; comunicação e flexibilidade.

Com o crescimento das redes e da Web, os sistemas distribuídos podem ser vistos e classificados de uma nova forma a partir destes sistemas.

Segundo (FREIRE, 2005), os sistemas distribuídos baseados na arquitetura Cliente/Servidor podem ser classificados em três grupos: rede centralizada; rede de computadores oferecendo serviços; servidores compartilhando serviços com clientes. As primeiras redes tinham uma visão centralizada do computador, tanto que, um único computador era o responsável por todos os serviços e recursos da rede. No modelo de rede de

computadores oferecendo serviços, percebe-se que há uma infinidade de serviços que podem ser adicionados, permitindo assim o crescimento do ambiente computacional de forma flexível e escalar, possibilitando a disponibilidade de novos recursos e serviços ao usuário com certa transparência. Já no modelo de servidores compartilhando serviços com clientes, alguns servidores de serviços compartilham o processamento com os computadores clientes quando solicitados.

O Mecanismo de Serialização de Objetos é o processo de codificação de um objeto Java em um *array* de *bytes*, assim como a desserialização dos objetos é o processo de instanciação de um objeto a partir de um *array* de *bytes*. Objetos podem ser salvos em um arquivo e depois reaberto, ou transmitidos via rede a uma outra aplicação Java, por exemplo, utilizando *stream* soquetes. Campos específicos podem ser marcados como transientes de modo que o serializador e desserializador percebam isto. Graças ao Mecanismo de Serialização de Objetos, os ambientes de troca de mensagem implementados em Java, tais como o JMPI (*Java Message Passing Interface*) e o mpiJava, alcançaram desempenho superior ao ambiente implementado em C, como por exemplo o MPICH (PRIMO,2005).

2.1 O Ambiente de Troca de Mensagens MPI

A Interface de Troca de Mensagens foi desenvolvida por um fórum de discussão com o objetivo de ser um padrão para a troca de dados entre diversas máquinas e, por isto, pode ser usada na programação de *clusters*, redes e até mesmo em máquinas massivamente paralelas, sendo juntamente com o PVM, as duas tecnologias mais populares.

O MPI é uma especificação de rotinas de comunicação para ambientes de troca de mensagens, que define como será a interface de programação e a semântica das rotinas de comunicação.

O maior objetivo do MPI é permitir que aplicações paralelas sejam portáveis entre diferentes plataformas, oferecendo também a possibilidade de executar, de modo transparente, em sistemas heterogêneos.

Ao utilizar o padrão MPI, uma aplicação é constituída por um ou mais processos que se comunicam, acionando-se funções para o envio e recebimento de mensagens entre os processos. Os elementos mais importantes nas implementações paralelas utilizando o padrão MPI é a comunicação de dados entre processos paralelos e o balanceamento de carga, sendo que, no padrão MPI, o número de processos normalmente é fixo.

2.1.1 Funções Básicas do MPI

As funções `MPI_INIT` e `MPI_FINALIZE` são usadas para iniciar e finalizar uma execução MPI. A `MPI_INIT` deve ser chamada antes de qualquer função MPI ser acionada por cada processador. Depois de a função `MPI_FINALIZE` é acionada, nenhuma outra função poderá ser chamada ou acessada.

A função `MPI_COMM_SIZE` determina o número de processos da execução concorrente e a função `MPI_COMM_RANK` os identifica, usando um número inteiro. Os processos pertencentes a um grupo, são identificados com um número inteiro precedido de um zero. As funções `MPI_SEND` e `MPI_RECV` são usadas para enviar e receber mensagens.

Todas as funções acima, exceto as duas primeiras, possuem um manipulador de comunicação como argumento. Esse manipulador identifica o grupo de processos e o contexto das operações que serão executadas. Os comunicadores fornecem o mecanismo para identificar um subconjunto de processos, assegurando que as mensagens não sejam confundidas.

Uma característica importante do MPI é que a programação de troca de mensagens não é determinística, ou seja, o MPI não garante que uma mensagem enviada por dois processos A e B ao processo C, cheguem na ordem em que foram enviadas. A responsabilidade da execução determinística no MPI é feita em código, utilizando algumas idéias dadas em seqüência.

A especificação da origem, no `MPI_RECV`, pode permitir o recebimento de mensagens vindas de um único processo específico, ou de qualquer processo. Ao permitir o recebimento de mensagens de qualquer processo, autoriza o recebimento de dados que tenha qualquer origem, porém, ao permitir o recebimento de mensagens de um único processo, permite que as mensagens cheguem na ordem do tempo em que foram enviadas.

Um mecanismo adicional para distinguir mensagens, que é preferível por reduzir a possibilidade de erros, é o uso do parâmetro "*tag*". O processo de envio deve associar uma "*tag*" (inteiro) a uma mensagem. O processo de recepção pode especificar que deseja receber mensagens com uma "*tag*" específica ou com qualquer "*anytag*".

Ao implementar algoritmos paralelos, é preciso usar operações coordenadas envolvendo múltiplos processos, sendo que o MPI fornece uma série de funções que executam operações comuns desse tipo, sendo elas:

- `BARRIER`: sincroniza todos os processos.
- `BROADCAST`: envia dados de um processo a todos os demais processos.
- `GATHER`: junta dados de todos os processos em um único processador.
- `SCATTERS`: distribui um conjunto de dados de um processo para todos os processos.
- `REDUCTION OPERATIONS`: soma, multiplicação, máximo, mínimo e operações lógicas.

A função `MPI_BARRIER()` é usada para sincronizar a execução de processos de grupo, ou seja, nenhum processo pode realizar qualquer instrução, até que todos os processos tenham passado por essa barreira.

As funções `MPI_BCAST()`, `MPI_GATHER()`, e `MPI_SCATTER()` são rotinas coletivas de movimentação de dados, nas quais todos os processos interagem com origens distintas para espalhar, juntar e distribuir os dados.

A função `MPI_BCAST()` implementa um processo de dispersão de dados do tipo um para todos, no qual um único processo origem envia um dado para todo os outros processos e cada processo recebe esse dado.

A função `MPI_GATHER()` implementa um processo de junção dos dados de todos os processos em um único processo. Todos os processos, incluindo a origem, enviam dados localizados na origem.

A função `MPI_SCATTER()` implementa uma operação de dispersão de um conjunto de dados para todos os processos.

Nas funções `MPI_BCAST()`, `MPI_GATHER()`, e `MPI_SCATTER()`, os três primeiros argumentos especificam o local e o tipo de dados a serem comunicados e o número de elementos a serem enviados para cada destino, os outros argumentos especificam o local, o tipo de resultado e o número de elementos a serem recebidos de cada origem.

As funções `MPI_REDUCE()` e `MPI_ALLREDUCE()` implementam operações de redução. Elas combinam valores fornecidos no *buffer* de entrada de cada processo, usando operações específicas, e retornam o valor combinado, ou para o *buffer* de saída de um único processo da origem, ou para o *buffer* de saída de todos os processos. Essas operações incluem o máximo e mínimo respectivamente com as funções `MPI_MAX` e `MPI_MIN`, soma e produto com as funções `MPI_SUN` e `MPI_PROD` e as operações lógicas.

2.1.2 Compilação e Execução

Os métodos de compilação e execução dependem da implementação utilizada, mas para simplificar, os principais passos para compilar e executar um programa MPI são:

1. Utilizar compiladores que reconheçam automaticamente as funções MPI, ou incluir manualmente as bibliotecas MPI no processo de compilação;
2. Verificar quais são os nós disponíveis no ambiente paralelo;
3. Fixar os parâmetros para o ambiente paralelo;
4. Executar o programa, que deverá ser executado em cada um dos nós escolhidos.

2.2 Ambientes Distribuídos em Java (JMPI e mpiJava)

JMPI (*Java Message Passing Interface*) é um projeto de propósito comercial da MPI *Software Technology*, Inc., feito a partir do trabalho de mestrado de Steven (MORIN, 2000) com o intuito de desenvolver um sistema de passagem de mensagem em ambientes paralelos utilizando a linguagem Java. O JMPI combina as vantagens da linguagem Java com as técnicas de passagem de mensagem entre processos paralelos em ambientes distribuídos, sendo uma implementação do MPI para multi-processamento de memória distribuída em Java.

JMPI é completamente escrito em Java e roda em qualquer máquina em que a JVM (*Java Virtual Machine*) é suportada. A implementação do MPI é feita com RMI (*Remote Method Invocation*), o qual é uma construção nativa em Java e permite que um método que está sendo executado localmente invoque um método de uma máquina remota, fazendo com isso que uma chamada a um método remoto tenha a mesma semântica que uma chamada a um

método local. Em conseqüência, os objetos podem ser passados como parâmetros e serem retornados como resultado da invocação dos métodos remotos.

Uma outra implementação do MPI em Java é o mpiJava. Este padrão é baseado em objetos, seguindo o modelo da especificação em C++, provendo acesso a uma implementação nativa do MPI através do JNI (*Java Native Interface*). O mpiJava possui uma hierarquia de classes organizada seguindo a hierarquia de classes do C++. Esta hierarquia está definida na especificação do MPI-2 e consiste em seis classes principais: `MPI`, `Group`, `Comm`, `Datatype`, `Status` e `Request`.

A classe MPI só possui membros estáticos e é responsável pela inicialização de constantes globais. A classe mais importante existente no pacote é a comunicador `Comm`. Toda a comunicação em mpiJava é realizada por objetos da classe `Comm` ou de suas subclasses.

Uma outra classe de extrema importância é a classe `Datatype`, responsável por descrever os tipos dos objetos para os *buffers* de passagem de mensagens, para poder enviar, receber ou qualquer outra função que envolva comunicação.

Segundo CARPENTER, 2002, para uma melhor desenvolvimento e performance do mpiJava foram realizadas algumas alterações na estrutura do MPI para assim atender melhor à estrutura da linguagem Java.

Algumas destas mudanças foram:

- O tipo básico `MPI.OBJECT` foi adicionado;
- As classes de exceções do MPI não são específicas como subclasses de `IOException`;
- Os construtores de tipos derivados como: `VECTOR`, `HVECTOR`, `INDEXED`, `HINDEXED` tornaram-se métodos não-estáticos.

Foi realizada também uma mudança nos métodos “destruidores” de MPI, deixando assim a utilização do sistema mais simples para o usuário, pois o mesmo não teria que se

preocupar com a limpeza dos métodos que estão em memória (CARPENTER, Getov, Judd, Skjellum and Fox, 2000).

Todas as classes de MPI estão dentro do pacote `mpi`, os nomes das classes e membros geralmente seguem as recomendações das convenções para os códigos Java da Sun. Isso por que normalmente estas convenções estão de acordo com as convenções de nomes para o padrão MPI-2, caso algum nome tenha de ser alterado o mesmo leva uma letra minúscula no inicio do nome da classe ou método (CARPENTER, Getov, Judd, Skjellum and Fox, 2000).

Algumas restrições de dados foram apresentadas, pois a JVM não implementa o conceito de um endereço físico ser passado para um dado, com isso o uso do `MPI_TYPE_STRUCT DATATYPE CONSTRUCTOR` foi reduzido tornando impossível enviar `DATATYPE` básicos misturados em uma única mensagem (CARPENTER, Getov, Judd, Skjellum and Fox, 2000).

Uma outra característica do mpiJava é com relação ao início do *buffer* de dados, onde a linguagem C e Fortran possuem dispositivos para tratar este tipo de problema, mas o Java não possui este recurso, porém prove a mesma flexibilidade associando o parâmetro a um elemento do *buffer* definindo a posição do elemento (CARPENTER, Getov, Judd, Skjellum and Fox, 2000).

O MPI não devolve mensagem de erro de forma explícita, deste modo as exceções do Java são utilizadas para apresentar os referidos erros (CARPENTER, Getov, Judd, Skjellum and Fox, 2000).

Segue abaixo um pequeno exemplo da implementação do mpiJava, simplificando o simples uso do mesmo. Os métodos básicos utilizados para a comunicação são o `SEND` e `RECEIVE` localizados dentro da classe `COMM`.

```
import mpi.* ;

  class Hello {
    static public void main(String[] args) {
      MPI.init(args) ;

      int myrank = MPI.COMM_WORLD.rank() ;
      if(myrank == 0) {
        char [] message = "Hello, there".toCharArray() ;
        MPI.COMM_WORLD.send(message, 0, message.length, MPI.CHAR, 1, 99) ;
      }
      else {
        char [] message = new char [20] ;
        MPI.COMM_WORLD.recv(message, 0, 20, MPI.CHAR, 0, 99) ;
        System.out.println("received:" + new String(message) + ":") ;
      }

      MPI.finish();
    }
  }
```

Figura 1 – Exemplo de implementação com o mpiJAVA

2.3 Recursos de Serialização em Java

Sendo Java uma linguagem de programação orientada a objetos, espera-se que ela ofereça alguma funcionalidade para a transferência de objetos. Essa funcionalidade é conhecida como Mecanismo de Serialização de Objetos.

O processo de serialização de objetos permite converter a representação de um objeto em memória para uma seqüência de *bytes* que pode então ser enviada para um `ObjectOutputStream` usando o método `writeObject()`, que por sua vez pode estar associado a um arquivo em disco ou a uma conexão de rede.

O processo inverso ao da serialização é conhecido como desserialização, e permite ler essa representação a partir de um `ObjetcInputStream` usando o método `readObject()`.

Objetos de classes para os quais são previstas serializações e desserializações devem implementar a interface `SERIALIZABLE`, do pacote `JAVA.IO`. Essa interface não específica nenhum método ou campo.

A serialização é um processo transitivo, ou seja, subclasses serializáveis de classes serializáveis são automaticamente incorporadas à representação serializada do objeto raiz, e para que o processo de desserialização possa ocorrer corretamente, todas as classes envolvidas no processo devem ter um construtor sem argumentos.

A serialização de um objeto deve ser implementada pela definição de um método `writeObject()`, com a seguinte sintaxe:

```
private void writeObject(ObjectOutputStream out) throws IOException
```

Figura 2 – Sintaxe do método writeObject()

Esse método realiza qualquer preparação necessária para a serialização e invoca o método `defaultWriteObject()` da classe `ObjectOutputStream`.

A desserialização do objeto é implementada pela definição de um método `readObject()`, que possui a seguinte sintaxe:

```
private void readObject(ObjectInputStream input) throws IOException,
ClassNotFoundException
```

Figura 3 – Sintaxe do método readObject()

Esse método invoca `defaultReadObject()` da classe `ObjectInputStream` e então realiza as ações complementares, aquelas realizadas em `writeObject()`, tais como leitura de dados adicionais e inicialização de variáveis.

Por exemplo, para serializar um objeto e enviá-lo de uma aplicação cliente para um servidor, é necessário a definição de três classes em Java. A classe DATA, que será o objeto a ser serializado é definida pela seguinte classe:

```
public class Data implements Serializable {
  private int i;
  private int tabela[];
  public Data(int x){
            i = x;
            tabela = new int[10];
            for(int j=0;j<10;j++)
                  tabela[j]=x;
            }
  public String toString() {
            return Integer.toString(i);
            }
  public void print_conteudo() {
            for(int j=0;j<10;j++)
                  System.out.print(tabela[j]+" " );
            System.out.println();
            }
}
```

Figura 4 – Exemplo de classe serializavel

A classe acima DATA ela instancia um vetor do tipo int de 10 posições e preenche todas as posições deste vetor com o valor passado como argumento pelo construtor da classe.

O cliente responsável por instanciar a classe DATA e enviá-la ao servidor é dado pela seguinte classe:

```
public class Client {
      public static void main (String args[]) {
      // args[0] <- hostname server
      if(args.length == 0){
            System.out.println("Digitar o IP servidor na linha de comando");
            System.exit(0);
            }

      String texto;
      Socket s = null;
      int serversocket = 6000;
        try{
            // 1o passo
            s = new Socket(args[0], serversocket);
            if(s == null)
                  System.out.println("NAO CONSEGUIU CRIAR O SOCKET");

            // 2o passo
            DataInputStream in = new DataInputStream( s.getInputStream());
            DataOutputStream out =
                  new DataOutputStream( s.getOutputStream());

            // Criar ObjecOutputStream e ObjectInputStream
            ObjectOutputStream obj_o = new ObjectOutputStream(out);
            ObjectInputStream obj_i = new ObjectInputStream(in);

            int contador=0;

             // 3o passo
            while(contador < 5){
                  contador++;
                    Data d = new Data(contador);
                  obj_o.writeObject(d);
                  obj_o.flush();
                  d.print_conteudo();
                  System.out.println("OBJETO ENVIADO");
                  String data = in.readUTF();
                  System.out.println("Received: "+ data);

            }

     }catch (UnknownHostException e){
                  System.out.println("Sock:"+e.getMessage());
          }catch (EOFException e){System.out.println("EOF:"+e.getMessage());
          }catch (IOException e){System.out.println("IO:"+e.getMessage());
      }finally {if(s!=null) try {s.close();}catch (IOException e) {
            System.out.println("close:"+e.getMessage());}}

 }
}
```

Figura 5 – Cliente Serialização

A classe cliente possui um laço de repetição que irá ser executado cinco vezes. A cada execução desse laço, é criado um objeto do tipo DATA, que é preenchido o vetor com o valor atual do contador do laço de repetição e é enviado pela rede até o servidor

A classe servidora da aplicação é definida a seguir:

```
import java.net.*;
import java.io.*;

public class Server{
public static void main(String args[]) throws  IOException,
ClassNotFoundException{
int serverPort = 6000;
DataInputStream in;
DataOutputStream out;
String resposta;
Socket clientSocket=null;
ServerSocket listenSocket=null;

listenSocket = new ServerSocket(serverPort);

while(true){
      System.out.println("A espera de conexao no socket "+serverPort);
      try{
      clientSocket = listenSocket.accept();

      in = new DataInputStream(clientSocket.getInputStream());
      out = new DataOutputStream(clientSocket.getOutputStream());

  // Criar ObjectinputStream e ObjectOutputStream
  ObjectOutputStream obj_o = new ObjectOutputStream(out);
  ObjectInputStream obj_i = new ObjectInputStream(in);
  Data d;

      while(true){
            d = (Data)obj_i.readObject();
            System.out.println("Recebeu: "+d);
        d.print_conteudo();
            out.writeUTF("OBJETO RECEBIDO");
            out.flush();
             }// while

  }catch (SocketException e){System.out.println("Socket:" + e);
  }catch(EOFException e){System.out.println("EOF:" + e);
  }catch(IOException e){System.out.println("IO:" + e);}
 }//while
}
}
```

Figura 6 – Exemplo de Servidor de Serialização

A classe servidora possui um laço de repetição que em cada execução recebe um objeto do tipo DATA enviado pelo cliente e imprimi o seu conteúdo.

Ao desenvolver essa aplicação para objeto de estudo, pode-se notar os três passos necessários para que ocorra a serialização de objetos e a sua respectiva transferência por uma conexão de rede. O primeiro passo consiste em criar o socket para estabelecer a conexão, o segundo passo está em obter os pontos de recepção e envio de dados e o último passo é a transferência do objeto pela rede.

CAPÍTULO 3 – IMPLEMENTAÇÃO

O resultado neste trabalho é a implementação do ambiente proposto, utilizando o conceito de serialização, sendo verificadas significativas diferenças de tempos nos resultados encontrados, com isso consolidando as expectativas de incremento de performance que a literatura indica para o uso do Mecanismo de Serialização (CARPENTER et all, 2000).

3.1 Ambiente de Serialização de Objetos

O ambiente desenvolvido responsável por tratar da transferência do objeto serializado pela rede foi construído baseando-se na arquitetura Cliente/Servidor e tem como principais classes Java: `SerializeImage.Java`, `Client.Java`, `Server.Java`.

Todas essas classes estão disponíveis no apêndice.

3.1.1 Classe Serializável

Para que fosse possível transportar os dados da imagem que está encapsulado no objeto Java do programa servidor para o programa cliente, foi necessário utilizar o mecanismo de serialização de objetos, ou seja, este objeto foi transformado em uma seqüência de *bytes* que posteriormente foi completamente restaurado para gerar novamente o objeto original.

A classe utilizada para isso foi a `SerializeImage.java`. Ela basicamente foi implementada para ser o "dado serializado", ou seja, um dos atributos desta classe é um *buffer* do tipo `byte[]`. Este *buffer* é utilizado para guardar os *bytes* da imagem a ser transferida do servidor para cliente.

Para que fosse possível a implementação da classe `SerializeImage.java` foi necessário fazer uso da palavra-chave *transient*, porque quando está sendo controlado a serialização do objeto, pode haver algum atributo específico que não seja necessário que o mecanismo de serialização do Java salve e restaure automaticamente. No caso desta classe, o atributo específico que não foi controlado na serialização foi o `FileInputStream`, utilizado para ler os *bytes* da imagem e armazená-los no *buffer*.

3.1.2 Módulo Servidor

A linguagem Java suporta a troca de *bytes* entre um cliente e servidor TCP através do estabelecimento de uma conexão entre eles. Essa conexão pode ser interpretada como ligações diretas entre dois processos através dos quais os *bytes* podem fluir entre os dois sentidos, sendo preciso identificar as extremidades dessa conexão tanto no processo cliente como no processo servidor. Essas extremidades, na aplicação, são representadas pelos sockets. Os soquetes são identificados por um endereço de rede e um número de porta, sendo que, nessa aplicação, poderia ter sido utilizada qualquer porta de comunicação, e a escolhida foi a 6000. A conexão entre o servidor e o cliente do ambiente dá-se através da instanciação de um objeto da classe `SOCKET` a partir do processo cliente. A aplicação cliente utiliza a porta 6000 do processo servidor para enviar solicitações de requisições de imagens e para receber retornos a suas solicitações. O processo servidor monitora constantemente a porta 6000, aguardando a chegada de solicitações de serviço, e quando alguma solicitação é recebida, a aplicação servidora executa o serviço e utiliza a conexão para enviar o retorno com os resultados do serviço. Uma vez estabelecida a conexão, os dados das imagens médicas podem fluir através dos *streams* a ela associado de forma serializada, utilizando os métodos `getInputStream` e `getOutputStream` para obter os pontos de recepção e envio dos dados.

O processo servidor (APÊNDICE A) do ambiente está preparado para responder às solicitações de conexões por parte dos clientes, permanecendo em estado de espera (*listening*) entre as solicitações. A classe Java utilizada para prover tal funcionalidade foi a `ServerSocket`.

Os *streams* (fluxo seqüencial de *bytes*) são os responsáveis para que os fluxos de *bytes* das imagens possam fluir, utilizando o conceito de cadeias unidirecionais de dados apenas de escrita ou apenas de leitura. Com isso a aplicação desenvolvida pode obter dados do inicio de um *stream* de entrada e enviar dados para o final de um *stream* de saída de dados, sempre seqüencialmente.

Streams em Java são suportados pelo pacote `java.io`. Para a leitura seqüencial do objeto serializado, utiliza-se o método `readObject()` da classe `ObjectInputStream`, obtido como retorno do método `getInputStream`. Similarmente, nesta aplicação, para a transferência do objeto serializado no sentido do soquete servidor para o outro extremo da conexão, utiliza-se o método `writeObject()` da classe `ObjectOutputStream`.

O servidor da aplicação obedece duas tarefas básicas para um servidor TCP/IP: permanecer em execução aguardando a chegada de requisições em alguma porta esperada, neste caso a 6000 e, responder à solicitação através de uma conexão estabelecida com o cliente em função da requisição. O método utilizado para implementar no servidor a espera bloqueada por uma solicitação foi o método `ACCEPT()` da classe `ServerSocket`.

Um dos principais problemas na execução de aplicações que se baseiam no modelo Cliente/Servidor está associado com o tempo de atendimento a uma solicitação pelo servidor. Caso o servidor seja um processo monolítico, ele estará indisponível para receber novas requisições enquanto a solicitação recebida não for completamente atendida. A solução para esse problema, adotada neste trabalho, foi estabelecer um processamento independente para o atendimento a cada solicitação ao servidor, liberando o mais rápido possível o servidor para

receber novas solicitações. Essa técnica foi implementada através do conceito de *Multithreaded*. A idéia implementada consiste do corpo principal de processamento permanecer em um laço eterno para aceitar solicitações na porta 6000 e criar um objeto *thread* para atender à solicitação recebida. Cada *thread* criada existe somente durante o tempo necessário para atender à solicitação do cliente.

Para que se pudesse oferecer o serviço de *multithreaded* foi necessário que a classe servidora implementasse a interface `java.lang.Runnable`, que define um método `run()`, os quais são utilizados para criar objetos da classe `java.lang.Thread`, com métodos `start()` e `stop()`, entre outros.

3.1.3 Módulo Cliente

O processo cliente (APÊNDICE B) do ambiente está concentrado nas classes: `Client.java`, `Login.java`, `CreateImage.java`, `MyCanvas.java`, `Register.java` e `ConnectionDatabase.java`. As principais funções deste processo cliente consistem em conectar-se ao servidor, através de IP/porta específicos, fazer solicitações (neste caso pedido de imagens que estão armazenadas no servidor), receber respostas destas solicitações e apresentá-las ao cliente.

A classe `Login.java` é utilizada para exibir a primeira tela do sistema, como mostrado na figura 7, em que o cliente pode efetuar o seu *login* para a entrada no sistema. O sistema permite que um cliente entre no sistema com a sua identificação e senha, ou caso este não possua estes dados, poderá efetuar seu cadastro, como mostrado na figura 8, pela classe `Register.java`, salvando os seus dados no banco de dados.

Figura 7 – Tela de login do sistema

Figura 8 – Tela de cadastro de novo usuário

A classe `ConnectionDatabase.java` é responsável por estabelecer a conexão com o banco de dados, e efetuar as consultas necessárias para a entrada do usuário no sistema.

3.1.3.1 Banco de Dados

O banco de dados, neste trabalho, foi utilizado para guardar os dados pertinentes do cliente, para que este possa ser identificado na entrada do sistema.

O banco de dados foi desenvolvido na ferramenta Microsoft Access, pelo fato de ser uma ferramenta simples de se utilizar.

Para utilizar o banco de dados na linguagem Java, a API utilizada foi a `java.sql` que possui um conjunto de classes que possibilita a realização das operações, tais como: abertura de tabelas, inclusão de dados, consultas, alterações e etc.

3.1.3.1.1 Carregamento do Driver

Para carregar o *driver*, o modelo utilizado foi o *driver* **JAVA/ODBC**, fazendo a conexão com o banco de dados através do comando:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Figura 9 – Comando para carregar o Driver

sendo "`sun.jdbc.odbc`" o pacote e, "`jdbcOdbcDriver`" a classe.

3.1.3.1.2 Realização da Conexão

A realização da conexão neste trabalho foi feita pela classe `DriverManager`, utilizando o comando mostrado na figura 10.

```
c = DriverManager.getConnection("jdbc:odbc:Autenticacao");
```

Figura 10 – Comando para realizar a conexão com o Banco de Dados

3.1.3.1.3 Consultas ao Banco de Dados

As operações feitas no banco de dados utilizam um objeto da classe `Statement`, como pode ser observado na figura abaixo.

```
Statement stm = connection.createStatement();
```

Figura 11 – Objeto da classe Statement

Todos os comandos SQL padrão são suportados pela linguagem Java, sendo utilizados os métodos `executeQuery()` e `executeUpdate()`.

Para obter informações do banco de dados, foi utilizado o método `executeQuery()` da classe `statement`, que executa consultas no banco de dados através da cláusula *select*, atribuindo os dados a um objeto do tipo `ResultSet`. As consultas, então, ficam armazenadas no objeto `ResultSet`, que possui uma estrutura semelhante a uma tabela, com os campos e respectivos dados selecionados.

O método `next()` foi utilizado para obter os dados armazenados no objeto `ResultSet`, que retornará *true* (verdadeiro) enquanto houver dados a serem lidos no objeto. A figura 12 mostra o trecho de código da consulta para efetuar o *login*.

```
try{
            stm =  connection.createStatement();
            ResultSet rs;
            rs = stm.executeQuery("Select * from LOGIN where USER =
'"+user+"';");
            if (!rs.next()){
                System.out.println("Usuário não Existe");
            }
            ResultSetMetaData rsmd = rs.getMetaData();
            if (password.equals(rs.getString(2))){
                ok = true;
            }
            else{
                ok = false;
                System.out.println("Senha Incompativel");
            }
        }catch (SQLException sqlException){
            sqlException.printStackTrace();
        }
```

Figura 12 – Parte do código de login do usuário ao sistema

Para inserir dados no banco de dados, foi utilizado o método `executeUpdate()` pertencente a classe `statement`.

3.1.3.2 Interface do Cliente

A classe `Client.java` é responsável por fazer a solicitação ao servidor, e mostrar ao usuário, da forma mais apropriada, a resposta do servidor.

A aplicação cliente foi desenvolvida na forma de Frame, e pode ser observada na figura 13.

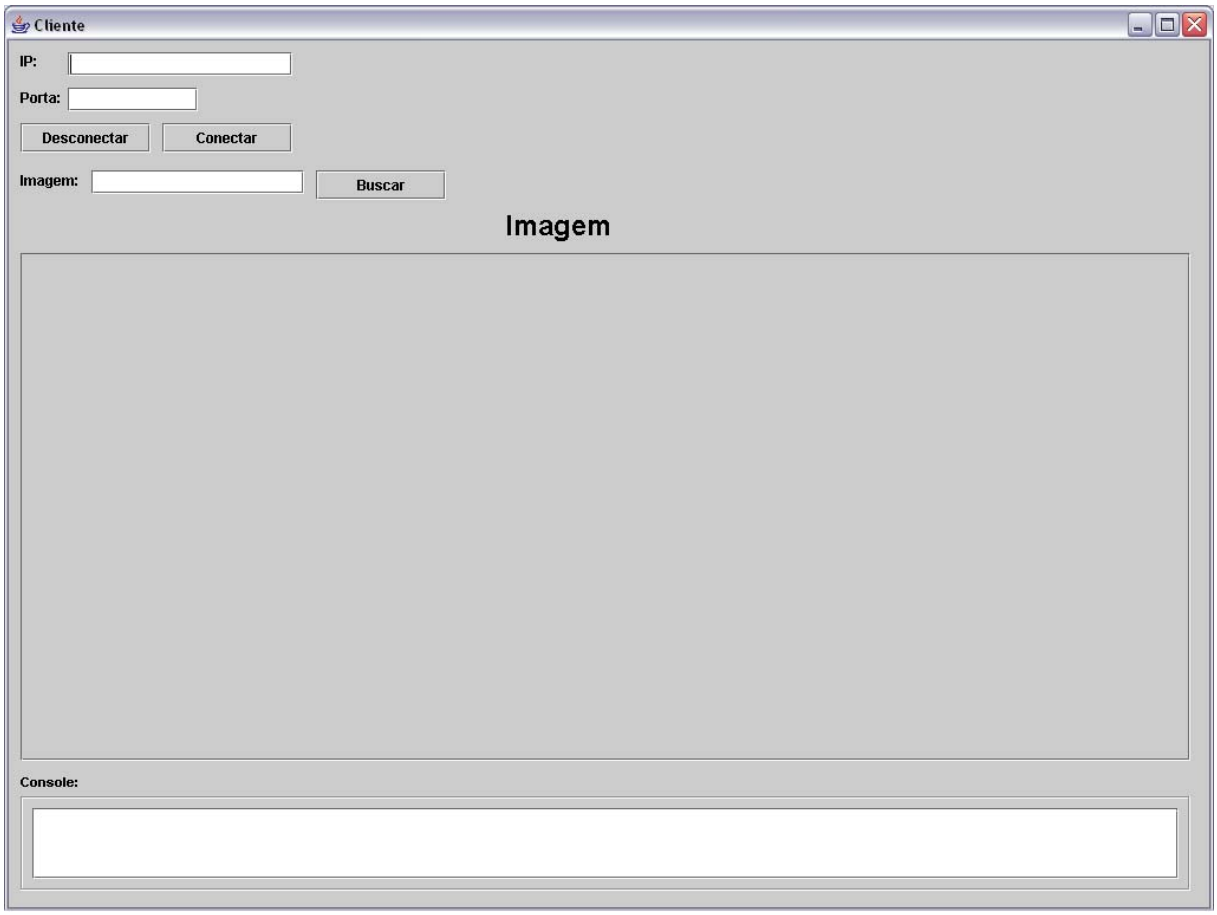

Figura 13 – Interface do Cliente

O cliente se conecta no servidor informando o IP e a porta, feito isso ele pode solicitar uma consulta ao servidor da aplicação informando o nome da imagem. Ao pressionar o `JButton` "Buscar" essas informações são transmitidas pela rede utilizando o protocolo TCP/IP ao servidor, este processa a requisição e responde ao cliente da forma mais apropriada. De posse da imagem solicitada, podendo ser em qualquer um dos formatos conhecidos (JPEG, PNG, TIF, etc) a interface do cliente possui um `CANVAS` que é utilizado para mostrar a imagem para o cliente.

3.1.3.3 Visualizando a Imagem

A imagem médica pode ser visualizada pelo cliente através do objeto `MyCanvas` localizado em sua interface. As classe `MyCanvas.java` é responsável em criar a tela de exibição para a imagem. Esta classe estende a classe `canvas`, e possuí métodos para retornar à imagem que está sendo exibida no `canvas` e carregar a imagem para a exibição. A classe implementada `CreateImage.java` é responsável por carregar a imagem em um objeto `IMAGE` para que possa ser adicionada no objeto da classe `MyCanvas`. Esta classe possui somente um único método responsável em criar a imagem.

No ponto de vista de imagens médicas, este trabalho tem uma continuidade em (TAMAE, 2005).

A interface do cliente exibindo a imagem pode ser observado na figura 14.

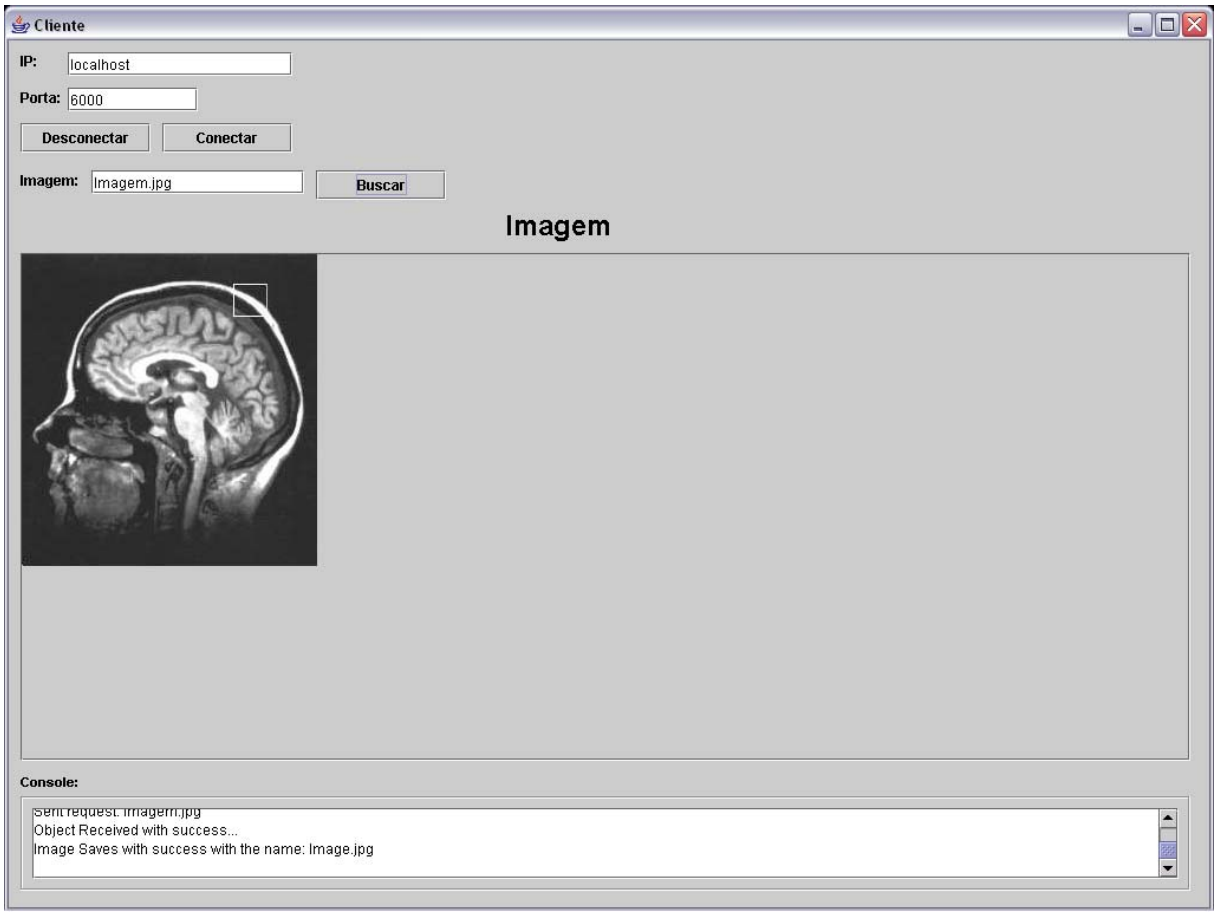

Figura 14 – Visualização da Imagem

3.2 Ambiente MPI

O ambiente responsável por tratar da troca de mensagem em MPI (PRIMO, 2005) foi desenvolvido em paralelo a este trabalho.

JMPI-PLUS foi o nome dado ao ambiente MPI desenvolvido, que tem como função atender requisições realizadas pelo ambiente de serialização desenvolvido, recuperando assim as imagens no banco de dados do servidor. Este ambiente é uma implementação do padrão MPI utilizando a linguagem Java. As imagens estão armazenadas no banco de dados, no formato DICOM. Este formato foi utilizado pelo fato de todas as modalidades médicas, tais como, Raio-X, Tomografia entre outras, adotarem este padrão para o transporte das imagens geradas.

CAPÍTULO 4 - RESULTADOS

Para provar o ganho de performance da serialização em relação à não serialização, foi desenvolvida uma aplicação para a transferência da imagem médica sem o uso da serialização, para que fosse possível comparar o desempenho dela em relação ao ambiente proposto. Foram realizados testes com 23 imagens no formato JPEG de tamanhos entre 460.647 e 26.299.956 bytes. A rede utilizada para testes foi a rede interna da instituição UNIVEM, com computadores Pentium IV – 2.80 GHZ, cache 256 KB, memória RAM 512 MB PC 2700, HD 40 GB 7200 RPM ATA 100, placa de rede 100/1000 Gb Intel Pro 1000, Switch: Gibabit Ethernet configurado como Full Duplex, velocidade da conexão: 100 Mbps.

Foi feito o transporte de cada imagem testada de forma serializada e não serializada, tendo sido executado o transporte em cada modo trinta vezes.

A métrica utilizada nas avaliações de performance é o tempo de transporte e a unidade utilizada é o milissegundo.

O gráfico da média dos tempos de cada imagem é mostrado na figura 15:

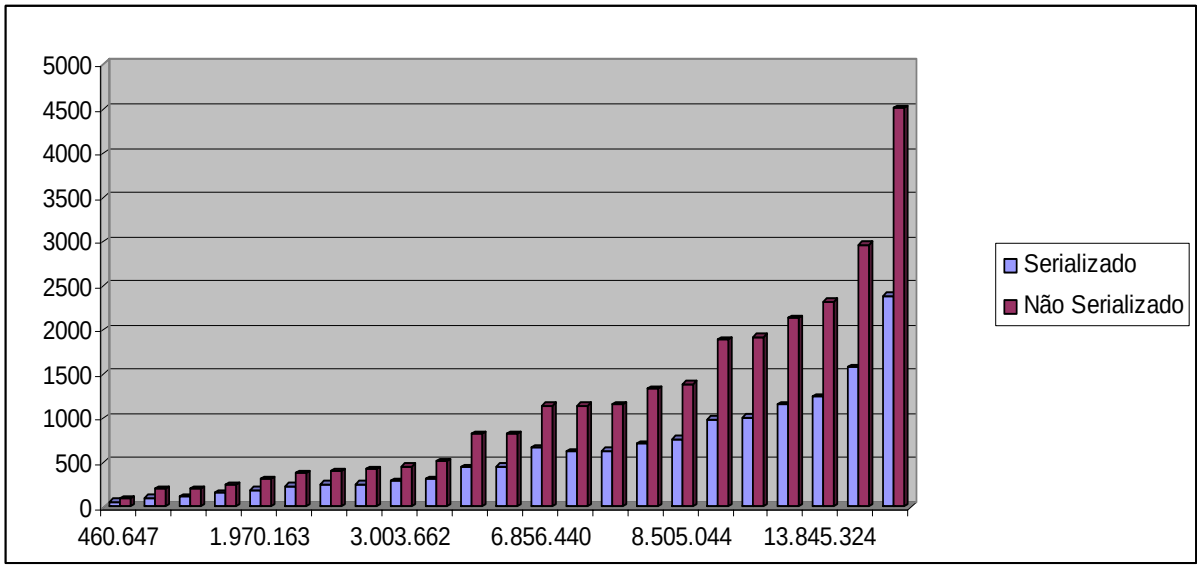

Figura 15 – Gráfico da média dos tempos de transporte

Como pode ser observado no gráfico, houve um ganho significativo da serialização em relação à não serialização, pois para todos os tamanhos de imagens utilizados no teste, a média da serialização sempre ficou abaixo da média da não serialização, podendo notar um padrão na média de 50%.

4.1 Teste de Hipótese

Para provar que o transporte utilizando a serialização obteve um ganho significativo sobre o transporte sem a serialização, foi feita uma análise estatística para verificar se as diferenças de desempenho das aplicações são estatisticamente significativas.

Para isso, a técnica estatística utilizada neste trabalho foi a denominada Teste de Hipóteses (FRANCISCO, 1995) (ACHCAR & RODRIGUES, 1995).

Essa técnica estatística é desenvolvida a partir de duas hipóteses, H_0 e H_1, que após realizar o cálculo dos testes, levará à aceitação de uma delas e à rejeição da outra. A hipótese H_0, geralmente, é formulada procurando-se "discordar" dos valores obtidos, já a hipótese H_1 é aquela que, geralmente, "concorda" com os valores obtidos no experimento. Assim, a hipótese H_0 é a negação da hipótese H_1.

A hipótese H_0 utilizada neste trabalho pode ser exemplificada da seguinte maneira: "o desempenho do transporte utilizando a serialização é melhor que o desempenho da mesma aplicação utilizando o transporte sem a serialização".

É necessária a definição dos seguintes testes de hipóteses neste trabalho para que seja possível realizar a análise estatística:

- Para amostras onde o tempo da Serialização < tempo da Não Serialização:

$$H_0: \mu Serialização \geq \mu NãoSerialização$$

$$H_1: \mu Serialização < \mu NãoSerialização$$

- Para amostras onde o tempo da Serialização > Não Serialização:

H_0: μSerialização ≤ μNãoSerialização

H_1: μSerialização > μNãoSerialização

Onde: μSerialização e μNãoSerialização são as médias dos tempos de transporte das imagens serializadas e não serializadas.

Supondo que os 30 tempos obtidos para cada média comparada apresentam uma distribuição normal, a estatística dos testes de hipóteses neste trabalho é dada como:

$$Z = \frac{\overline{X}serializa - \overline{X}\tilde{n}serializa}{\sqrt{\frac{S^2 serializa}{Nserializa} + \frac{S^2 \tilde{n}serializa}{N\tilde{n}serializa}}}$$

Figura 16 – Formula estatística do Teste de Hipóteses

onde: $\overline{X}serializa$ e $\overline{X}\tilde{n}serializa$ são as médias amostrais dos tempos obtidos; $S^2 serializa$ e $S^2 \tilde{n}serializa$ representam o desvio padrão amostral; $Nserializa$ e $N\tilde{n}serializa$ representam o tamanho das amostras (neste trabalho 30).

Para a probabilidade de estar correto 99% das vezes que a análise estatística for feita, a hipótese H_0 é rejeitada caso $Z \leq -2.57$, ou então, $Z \geq 2.57$. Esse valor de Z é fornecido pela Tabela de Distribuição Normalizada (ACHCAR & RODRIGUES, 1995).

Foram utilizadas nove amostras das 23 imagens testadas. Essas 9 amostras estão divididas em três classes: pequena, média, grande. Cada uma dessas classes possuindo três imagens. Uma imagem grande neste trabalho é considerada tendo 26.299.956 bytes.

Os gráficos apresentados nas figuras que seguem apresentam os valores obtidos, bem como as tabelas apresentam os testes de hipótese realizados.

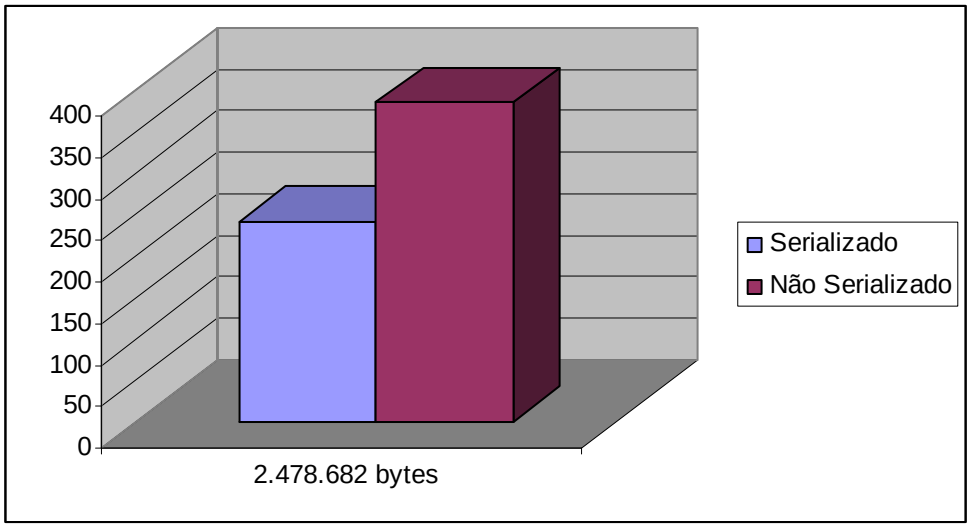

Figura 17 – Média da serialização em comparação com a não serialização (amostra 1)

	SERIALIZADO	NÃO SERIALIZADO
Média	240,6	384,5666667
Desvio Padrão	12,75390407	31,95976278
Hipótese α = 0,01	-22,91557525	

Tabela 1 – Valores calculados para o desempenho obtido com o transporte Serializado e o Não Serializado (amostra 1)

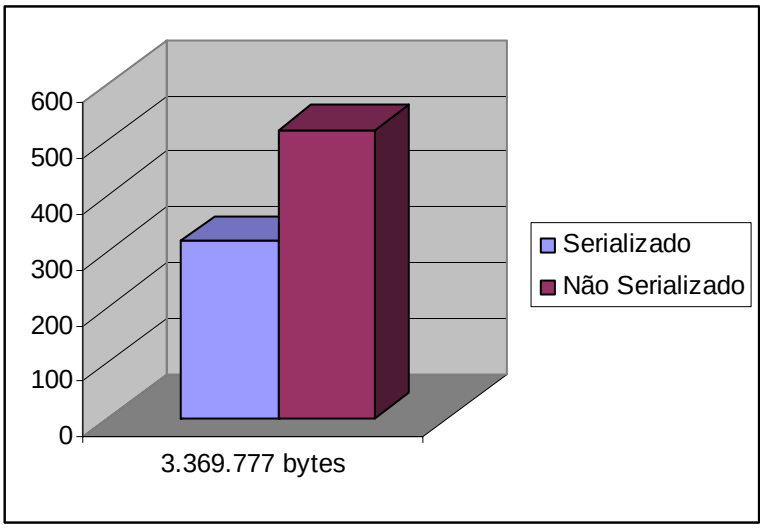

Figura 18 – Média da serialização em comparação com a não serialização (amostra 2)

	SERIALIZADO	NÃO SERIALIZADO
Média	317,2666667	516,8666666
Desvio Padrão	7,234654787	0,379049022
Hipótese α = 0,01	-150,9065467	

Tabela 2 – Valores calculados para o desempenho obtido com o transporte Serializado e o Não Serializado (amostra 2)

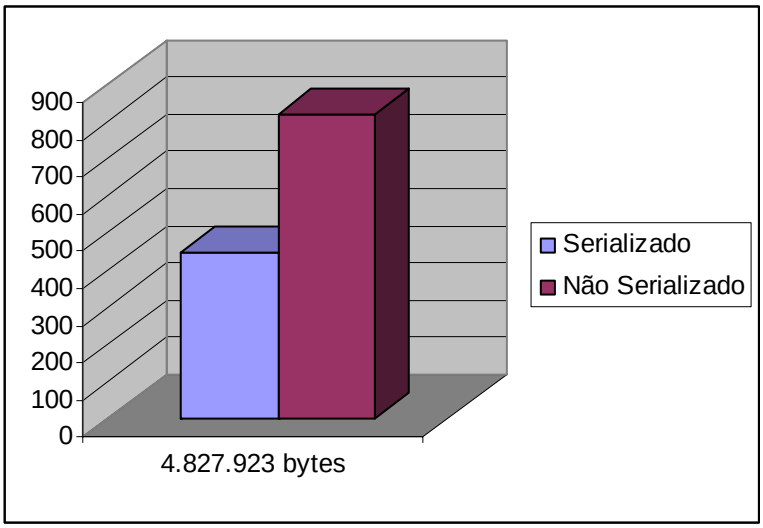

Figura 19 – Média da serialização em comparação com a não serialização (amostra 3)

	SERIALIZADO	NÃO SERIALIZADO
Média	447,9666667	819,3
Desvio Padrão	8,50348137	25,35832856
Hipótese α = 0,01	-76,04384212	

Tabela 3 – Valores calculados para o desempenho obtido com o transporte Serializado e o Não Serializado (amostra 3)

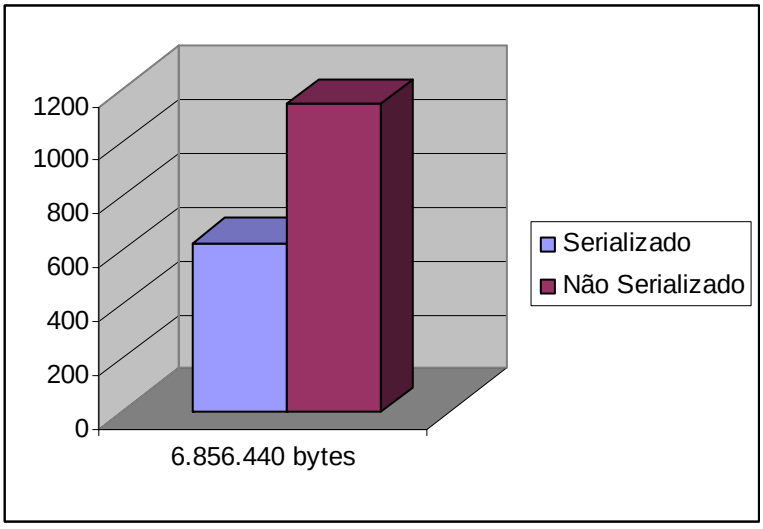

Figura 20 – Média da serialização em comparação com a não serialização (amostra 4)

	SERIALIZADO	NÃO SERIALIZADO
Média	625,4666667	1143,533333
Desvio Padrão	11,88372789	0,507416263
Hipótese α = 0,01	-238,5602304	

Tabela 4 – Valores calculados para o desempenho obtido com o transporte Serializado e o Não Serializado (amostra 4)

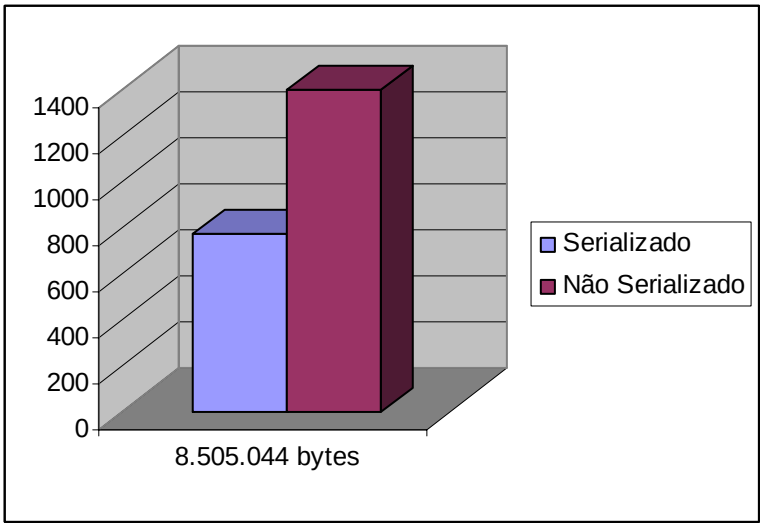

Figura 21 – Média da serialização em comparação com a não serialização (amostra 5)

	SERIALIZADO	NÃO SERIALIZADO
Média	769,2666667	1393,233333
Desvio Padrão	8,819779235	7,876999837
Hipótese α = 0,01	-289,0101461	

Tabela 5 – Valores calculados para o desempenho obtido com o transporte Serializado e o Não Serializado (amostra 5)

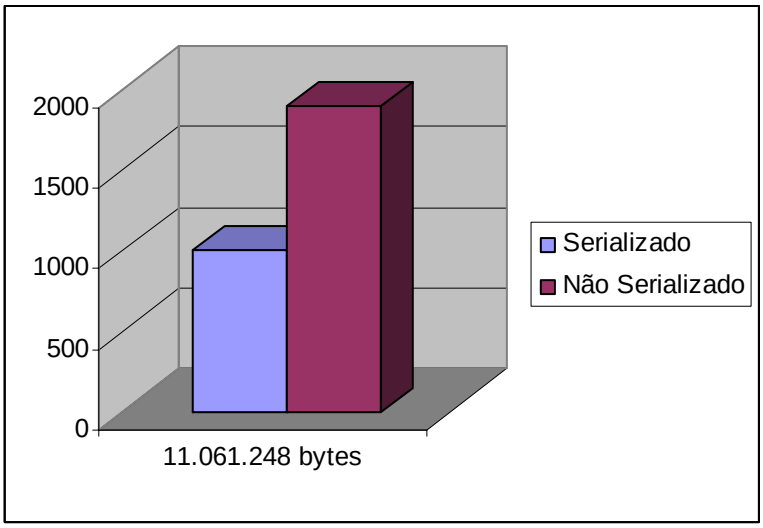

Figura 22 – Média da serialização em comparação com a não serialização (amostra 6)

	SERIALIZADO	NÃO SERIALIZADO
Média	995,3333333	1888,633333
Desvio Padrão	9,378380617	88,2420196
Hipótese α = 0,01	-55,13704515	

Tabela 6 – Valores calculados para o desempenho obtido com o transporte Serializado e o Não Serializado (amostra 6)

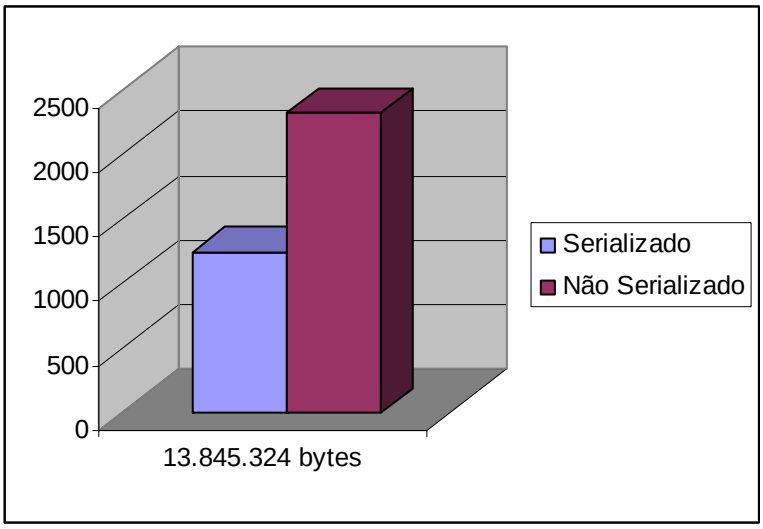

Figura 23 – Média da serialização em comparação com a não serialização (amostra 7)

	SERIALIZADO	NÃO SERIALIZADO
Média	1241,066667	2318,633333
Desvio Padrão	13,99490876	7,141347952
Hipótese α = 0,01	-375,6493561	

Tabela 7 – Valores calculados para o desempenho obtido com o transporte Serializado e o Não Serializado (amostra 7)

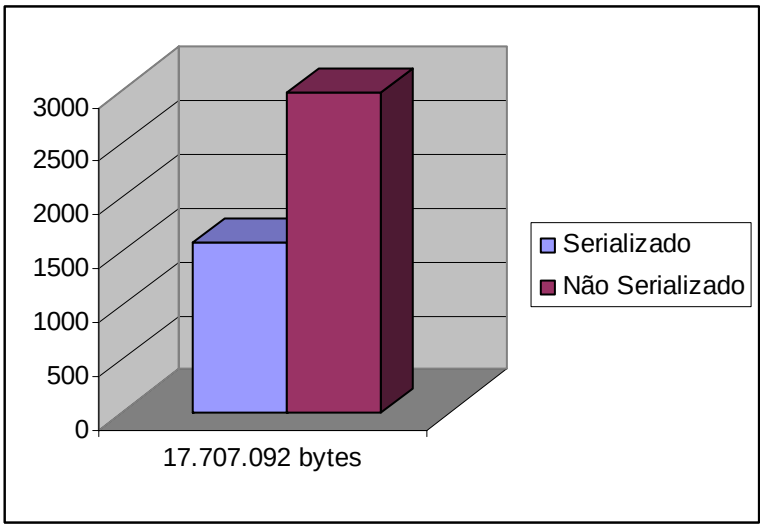

Figura 24 – Média da serialização em comparação com a não serialização (amostra 8)

	SERIALIZADO	NÃO SERIALIZADO
Média	1573,466667	2964,6
Desvio Padrão	10,29138687	5,366563146
Hipótese α = 0,01	-656,4856005	

Tabela 8 – Valores calculados para o desempenho obtido com o transporte Serializado e o Não Serializado (amostra 8)

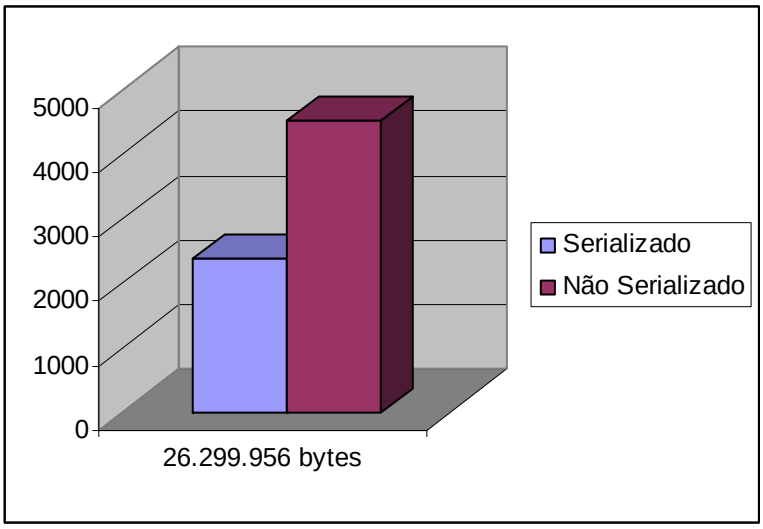

Figura 25 – Média da serialização em comparação com a não serialização (amostra 9)

	SERIALIZADO	NÃO SERIALIZADO
Média	2383,9	4507,766667
Desvio Padrão	56,50288366	10,90296807
Hipótese α = 0,01	-202,1523237	

Tabela 9 – Valores calculados para o desempenho obtido com o transporte Serializado e o Não Serializado (amostra 9)

CAPÍTULO 5 – CONCLUSÕES

Percebeu-se com o estudo da revisão bibliográfica que a linguagem Java se apresentou como uma escolha natural para o desenvolvimento de sistemas distribuídos, em função das facilidades para a criação e sincronização de *threads*, suporte à comunicação em rede, independência de plataforma, segurança e robustez. Ela apresentou também facilidades nativas que não são encontradas em outras linguagens. No entanto, percebeu-se também como essas facilidades impactam negativamente no seu desempenho.

Notou-se que a serialização de objetos é um procedimento utilizado em diversas circunstâncias, porque dispositivos que recebem ou devolvem dados em fluxos contínuos não comportam naturalmente os objetos, que, em sua forma mais genérica, possuem estruturas não lineares. Mas também há situações importantes em que os objetos precisam estar em dispositivos seriais, como no caso deste projeto, que utiliza a transmissão de objetos através da rede.

Os valores apresentados no Teste de Hipótese (tabelas 1 a 9), validam a implementação e demonstram que transporte utilizando o Mecanismo de Serialização de Objetos permite ganhos reais de desempenho estatisticamente significativos, quando comparados com o transporte realizado sem o uso do Mecanismo de Serialização, em valores proporcionais, em ordem crescente, na medida em que os tamanhos das imagens aumentam, chegando próximo de 50% de ganho para imagens de 26 Mbytes.

O trabalho teve um andamento considerado normal, com algumas etapas importantes, como a validação dos resultados. Trabalhos futuros poderiam consistir no acoplamento deste trabalho com dois outros trabalhos de mestrado (PRIMO, 2005) (TAMAE, 2005).

Muitos conceitos estudados no curso, tanto de aspectos práticos de programação como teóricos foram aprofundados no decorrer do projeto, contribuindo significativamente para a formação acadêmica.

REFERÊNCIAS

(ACHCAR & RODRIGUES, 1995) - Achcar, J.A., Rodrigues, J. Introdução à Estatística para Ciências e Tecnologia. ICMSC-USP, São Carlos - Apostila de Consulta, 1995.

(BEG, 1994) – A., Geist, A., Dongara, J., Jiang, W., Manchek, R., Sunderam, V., "PVM: Parallel Virtual Machine: A User's Guide and Tutorial for Networked Parallel Computing", The MIT Press, 1994.

(CARPENTER, Fox, Sung & Sang, 1999) - CARPENTER, Bryan, Geoffrey Fox, Sung Hoon Ko and Sang Lim. Object Serialization for Marshalling Data in a Java Interface to MPI. Proceedings of the ACM 1999 - Conference on Java Grande, San Francisco, CA, USA, 1999.

(CARPENTER, Getov, Judd, Skjellum and Fox, 1998) - Carpenter, B., Getov, V., JUDD, G., Skjellum, T. and Fox, G. - MPI for Java - Position Document and Draft API Specification, November 1998 – Available in: http://www.npac.syr.edu/projects/pcrc/mpiJava , access in: june 2004.

(CARPENTER, Getov, Judd, Skjellum and Fox, 2000) - Carpenter, B., Getov, V., JUDD, G., Skjellum, T. and Fox, G. - MPJ: MPI-like Message Passing for Java. *Concurrency: Practice and Experience*, v. 12, n. 11., 2000.

(DEITEL, 2003) - Deitel, H.M. & Deitel, P.J. - Java Como Programar - 4a edição, Porto Alegre, BookMan, 2003.

(FERRARI, 1998) - Ferrari, A.J. - JPVM: Network parallel computing in Java, In ACM 1998 Workshop on Java for High-Performance Network Computing. Palo Alto, February 1998, Concurrency: Practice and Experience, 1998.

(FRANCISCO, 1995) - Francisco, W. Estatística Básica, Unimep. Piracicaba, 2 ª Edição, 1995.

(FREIRE, 2005) - FREIRE, A.L., Sistemas Distribuídos em ambientes gráficos da Web Semântica, qualificação de Mestrado, Fundação Eurípides de Marília – UNIVEM, 2005.

(MORIN, 2000) – MORIN, S. R. - JMPI: Implementing the Message Passing Interface Standard in Java, University of Massachusetts, graduate degree in Master of Sience in Electrical and Computer Enginnering, september 2000, disponível em: http://www.ecs.umass.edu/ece/realtime/ publications/steve-thesis.pdf.

(MPI, 1994) - The Message Passing Interface (MPI) standard- The Message Passing Interface (MPI) Standard, version 1.0 of May 1994 - Available in: http://www-unix.mcs.anl.gov/mpi/index.html , access on: june of 2004.

(MPICH, 1996) - MPICH-A Portable Implementation of MPI, manual users organized by Mathematics and Computer Science Division,Argonne National Laboratory, site Available in: http://www-unix.mcs.anl.gov/mpi/mpich/ , acess on: june of 2004.

(mpiJava, 1999) - A HP Java Project - mpiJava Home Page, 1st mpiJava API Specification, Available on: http://www.hpjava.org/mpiJava.html, access on: june 2004.

(PACHECO, 1997) - PACHECO, Peter S.: Parallel programming with MPI. San Francisco – Morgan Kaufmann Publishers, CA, USA, 1997.

(PRIMO, 2005) - PRIMO, André L. G., Serialização e Performance em Ambientes Distribuídos usando MPI, 2005. Grau: Dissertação (Mestrado em Ciência da Computação), Centro Universitário Eurípides de Marília/UNIVEM, Marília, 2005.

(TAMAE, 2004) - TAMAE, Rodrigo Y., Sistema de Processamento Distribuído de Imagens Médicas com XML, 2005. Grau: Dissertação (Mestrado em Ciência da Computação), Centro Universitário Eurípides de Marília/UNIVEM, Marília, 2005.

APÊNDICE

APÊNDICE A – Classe Server

```
/**
 *
 * @author  vasco
 */

import java.net.*;
import java.io.*;
import java.awt.Image;
import javax.imageio.ImageIO;

public class Server implements Runnable{
    int serverPort = 6000;
    DataInputStream in;
    DataOutputStream out;
    String resposta;
    Socket request = null;
    ServerSocket server=null;
    public Server() {
        startServer();
    }
    public Server(Socket request) {
        this.request = request;
    }

    public void startServer(){
        try{
            server = new ServerSocket(serverPort);

            while (true) {
                System.out.println("Waiting for connection in the port
6000...\n");
                Socket socket = server.accept();
                System.out.println("Request from client received....");
                System.out.println("Host Name:
"+socket.getInetAddress().getHostName());
                System.out.println("IP: "+socket.getInetAddress());
                System.out.println("Port: "+socket.getPort());
                System.out.println("Waiting Request...");

                Server handler = new Server(socket);
                Thread thread = new Thread(handler);
                thread.start();
            }
        }catch(Exception e) {
            e.getMessage();
        }

    }
    public void run() {
        try {
            handleRequest();
        } catch (IOException ioex) {
```

```
            System.out.println("Error occurred while trying to service
request.");
            System.out.println("Server stopped.");
            ioex.printStackTrace();
            System.exit(0);
        }
    }

    public void handleRequest() throws IOException {

        try{

            System.out.println("New received Connection");
            in = new DataInputStream(request.getInputStream());
            out = new DataOutputStream(request.getOutputStream());


            ObjectOutputStream obj_o = new ObjectOutputStream(out);
            ObjectInputStream obj_i = new ObjectInputStream(in);

            String fileName = in.readUTF();
            System.out.println("Requested image: "+fileName);
            SerializeImage image = new SerializeImage(fileName);
            long begin = System.currentTimeMillis();
            obj_o.writeObject(image);
            obj_o.flush();
            long end = System.currentTimeMillis();
            System.out.println("Sent object");

            String message = in.readUTF();
            if(message.equals("Object Received with success"))
                System.out.println("Message of confirmation of the
delivery: "+ message);
            else {
                System.out.println("I not object received by the client");
                System.exit(0);
            }
            //long end = System.currentTimeMillis();
            System.out.println("Time in milliseconds: "+(end-begin));

        }catch (UnknownHostException e){
            System.out.println("Sock:"+e.getMessage());
        }catch (EOFException e){System.out.println("EOF:"+e.getMessage());
        }catch (IOException e){System.out.println("IO:"+e.getMessage());

        }
    }

    public static void main(String args[]) throws
IOException,ClassNotFoundException{
        Server server = new Server();
    }
}
```

APÊNDICE B – Classe Client

```
/**
 *
 * @author vasco
 */

import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import java.io.*;
import java.net.*;
import javax.swing.*;
import javax.swing.border.*;

public class Client extends javax.swing.JFrame {

      private JButton jButton1;
      private JButton jButton10;
      private JButton jButton2;
      private JLabel jLabel1;
      private JLabel jLabel2;
      private JLabel jLabel3;
      private JLabel jLabel4;
      private JLabel jLabel5;
      private JPanel jPanel1;
      private JScrollPane jScrollPane1;
      private JScrollPane jScrollPane2;
      private JTextArea jTextArea1;
      private JTextField jTextField1;
      private JTextField jTextField2;
      private JTextField jTextField3;
      private java.awt.Panel panel1;
      private MyCanvas canvas1 = new MyCanvas(null);
      private MyCanvas canvas2 = new MyCanvas(null);
      private int flag = 0;
      private static FileOutputStream fout;
      private static File fp;
      SerializeImage d;
      private int serversocket = 6000;
      private Socket s;
      private DataInputStream in;
      private DataOutputStream out;
      private ObjectOutputStream obj_o;
      private ObjectInputStream obj_i;
      private CreateImage gi;

      public Client() {
            iniciaGUI();
      }

      private void iniciaGUI() {
            jLabel1 = new JLabel();
            jLabel2 = new JLabel();
            jTextField1 = new JTextField();
            jTextField2 = new JTextField();
            jButton1 = new JButton();
            jButton2 = new JButton();
            jLabel3 = new JLabel();
            jTextField3 = new JTextField();
```

```
    jLabel4 = new JLabel();
    jButton10 = new JButton();
    jPanel1 = new JPanel();
    jScrollPane1 = new JScrollPane();
    jTextArea1 = new JTextArea();
    jLabel5 = new JLabel();
    jScrollPane2 = new JScrollPane();
    panel1 = new Panel();

    getContentPane().setLayout(null);

    setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
    setTitle("Cliente");
    addWindowListener(new WindowAdapter() {
          public void windowClosing(WindowEvent evt) {
                exitForm(evt);
          }
    });

    jLabel1.setFont(new Font("MS Sans Serif", 1, 12));
    jLabel1.setText("IP:");
    getContentPane().add(jLabel1);
    jLabel1.setBounds(10, 10, 34, 15);

    jLabel2.setFont(new Font("MS Sans Serif", 1, 12));
    jLabel2.setText("Porta:");
    getContentPane().add(jLabel2);
    jLabel2.setBounds(10, 40, 40, 16);

    getContentPane().add(jTextField1);
    jTextField1.setBounds(50, 10, 190, 20);

    getContentPane().add(jTextField2);
    jTextField2.setBounds(50, 40, 110, 20);

    jButton1.setFont(new Font("MS Sans Serif", 1, 12));
jButton1.setText("Conectar");
jButton1.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        jButton1ActionPerformed(evt);
    }
});

getContentPane().add(jButton1);
jButton1.setBounds(130, 70, 110, 25);

    jButton2.setFont(new Font("MS Sans Serif", 1, 12));
    jButton2.setText("Desconectar");
    getContentPane().add(jButton2);
    jButton2.setBounds(10, 70, 110, 25);

    jLabel3.setFont(new Font("MS Sans Serif", 1, 12));
    jLabel3.setText("Imagem:");
    getContentPane().add(jLabel3);
    jLabel3.setBounds(10, 110, 60, 16);

    getContentPane().add(jTextField3);
    jTextField3.setBounds(70, 110, 180, 20);

    jLabel4.setFont(new Font("MS Sans Serif", 1, 24));
```

```
            jLabel4.setText("Imagem");
            getContentPane().add(jLabel4);
            jLabel4.setBounds(420, 140, 220, 32);

            jButton10.setFont(new Font("MS Sans Serif", 1, 12));
            jButton10.setText("Buscar");
            jButton10.addActionListener(new ActionListener() {
                  public void actionPerformed(ActionEvent evt) {
                        jButton10ActionPerformed(evt);
                  }
            });

            getContentPane().add(jButton10);
            jButton10.setBounds(260, 110, 110, 25);

            jPanel1.setLayout(null);

            jPanel1.setBorder(new EtchedBorder());
            jScrollPane1.setViewportView(jTextArea1);

            jPanel1.add(jScrollPane1);
            jScrollPane1.setBounds(10, 10, 970, 60);

            getContentPane().add(jPanel1);
            jPanel1.setBounds(10, 640, 990, 80);

            jLabel5.setFont(new Font("MS Sans Serif", 1, 11));
            jLabel5.setText("Console:");
            getContentPane().add(jLabel5);
            jLabel5.setBounds(10, 620, 80, 15);

            jScrollPane2.setViewportView(panel1);

            getContentPane().add(jScrollPane2);
            jScrollPane2.setBounds(10, 180, 990, 430);

            Dimension screenSize =
Toolkit.getDefaultToolkit().getScreenSize();
            setBounds((screenSize.width - 1024) / 2, (screenSize.height -
768) / 2,
                        1024, 768);
      }

      private void jButton10ActionPerformed(ActionEvent evt) {
            String message = jTextField3.getText();
            try {
                  out.writeUTF(message);
                  out.flush();
                  jTextArea1.append("Sent request: " + message + "\n");

                  d = (SerializeImage) obj_i.readObject();
                  out.writeUTF("Object Received with success");
                  out.flush();
                  jTextArea1.append("Object Received with success...\n");

                  fp = new File("Image.jpg");
                  fout = new FileOutputStream(fp);
                  fout.write(d.bdata, 0, d.n);
                  fout.close();
```

```
                  jTextArea1
                              .append("Image Saves with success with the
name: Image.jpg\n");

                  panel1.add(canvas1);
                  gi = CreateImage.createImage("Image.jpg");
                  canvas1.setImage(gi);

            } catch (EOFException e) {
                  System.out.println("EOF:" + e.getMessage());
                  jTextArea1.append("Mistake: \n EOF: " + e.getMessage() +
"\n");
            } catch (Exception e) {
                  System.out.println("IO:" + e.getMessage());
                  jTextArea1.append("Mistake: \n IO: " + e.getMessage() +
"\n");
            } finally {
                  if (s != null)
                        try {
                              s.close();
                        } catch (IOException e) {
                              System.out.println("close:" + e.getMessage());
                        }
            }
      }

      private void jButton1ActionPerformed(ActionEvent evt) {
            String host = jTextField1.getText();
            int port = Integer.parseInt(jTextField2.getText());
            try {
                  s = new Socket(host, 6000);
                  if (s == null) {
                        System.out.println("Connection no established");
                        jTextArea1.append("Connection no
established...\n");
                  } else {
                        jTextArea1.append("Established connection\n");
                        jTextArea1.append("Host: " + jTextField1.getText()
+ "\n");
                        jTextArea1.append("Port: " + jTextField2.getText()
+ "\n\n");
                  }
                  in = new DataInputStream(s.getInputStream());
                  out = new DataOutputStream(s.getOutputStream());
                  obj_o = new ObjectOutputStream(out);
                  obj_i = new ObjectInputStream(in);
            } catch (Exception e) {
                  System.out.println("Sock:" + e.getMessage());
            }
      }

      private void exitForm(WindowEvent evt) {
            System.exit(0);
      }

      public static void main(String args[]) {
            new Client().show();
      }

}
```

APÊNDICE C – Classe SerializeImage

```
/**
 *
 * @author  vasco
 */
import java.io.*;
import java.net.*;
import java.awt.*;
import javax.imageio.ImageIO;

public class SerializeImage implements Serializable{
    protected File file;
    protected transient FileInputStream fins;
    protected byte[] bdata;
    protected int n;
    protected long bsize;
    protected int Bsize;
    public SerializeImage(String fileName) {

        try{
            file = new File(fileName);
            fins = new FileInputStream(file);
            long fsize = file.length();

            bsize = file.length();
            Bsize = (int) bsize;
            bdata = new byte[Bsize];
            n = fins.read(bdata, 0, Bsize);
        }catch(Exception e){}
    }
}
```

APÊNDICE D – Classe Login

```
/**
 *
 * @author  vasco
 */

import javax.swing.*;

public class Login extends JFrame {

      private JButton jButton1;
    private JButton jButton2;
    private JButton jButton3;
    private JButton jButton4;
    private JLabel jLabel1;
    private JLabel jLabel2;
    private JTextField txtId;
    private JPasswordField txtPass;

    public Login() {
        initComponents();
    }

    private void initComponents() {
        jLabel1 = new JLabel();
        jLabel2 = new JLabel();
        txtId = new JTextField();
        jButton1 = new JButton();
        jButton2 = new JButton();
        jButton3 = new JButton();
        jButton4 = new JButton();
        txtPass = new JPasswordField();


        getContentPane().setLayout(null);

        setTitle("Login");
        addWindowListener(new java.awt.event.WindowAdapter() {
            public void windowClosing(java.awt.event.WindowEvent evt) {
                exitForm(evt);
            }
        });

        jLabel1.setFont(new java.awt.Font("MS Sans Serif", 1, 12));
        jLabel1.setText("Usu\u00e1rio:");
        getContentPane().add(jLabel1);
        jLabel1.setBounds(10, 10, 60, 16);

        jLabel2.setFont(new java.awt.Font("MS Sans Serif", 1, 12));
        jLabel2.setText("Senha:");
        getContentPane().add(jLabel2);
        jLabel2.setBounds(10, 40, 80, 16);

        getContentPane().add(txtId);
        txtId.setBounds(90, 10, 150, 20);

        jButton1.setText("Cancelar");
        jButton1.addActionListener(new java.awt.event.ActionListener() {
```

```
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                jButton1ActionPerformed(evt);
            }
        });

        getContentPane().add(jButton1);
        jButton1.setBounds(20, 120, 100, 23);

        jButton2.setText("Limpar");
        jButton2.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                jButton2ActionPerformed(evt);
            }
        });

        getContentPane().add(jButton2);
        jButton2.setBounds(130, 120, 100, 23);

        jButton3.setText("OK");
        jButton3.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                jButton3ActionPerformed(evt);
            }
        });

        getContentPane().add(jButton3);
        jButton3.setBounds(130, 80, 100, 23);

        jButton4.setText("Registrar");
        jButton4.setToolTipText("To register new user");
        jButton4.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                jButton4ActionPerformed(evt);
            }
        });

        getContentPane().add(jButton4);
        jButton4.setBounds(20, 80, 100, 23);

        getContentPane().add(txtPass);
        txtPass.setBounds(90, 40, 150, 22);

        java.awt.Dimension screenSize =
java.awt.Toolkit.getDefaultToolkit().getScreenSize();
        setBounds((screenSize.width-256)/2, (screenSize.height-197)/2, 256,
197);
    }

    private void jButton4ActionPerformed(java.awt.event.ActionEvent evt) {
        Register register = new Register();
        register.show();
    }

    private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
        txtId.setText("");
        txtPass.setText("");
    }

    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
        System.exit(0);
```

```
    }

    private void jButton3ActionPerformed(java.awt.event.ActionEvent evt) {
        boolean ok;
        ConnectionDatabase bc = new ConnectionDatabase();
        ok = bc.consulta(txtId.getText(),txtPass.getText());

        if(ok){
            try {
                Client login = new Client();
                login.setTitle("Login");
                login.show();
            }catch(Exception e){
                e.printStackTrace();
            }

            this.dispose();
        }else{
            JOptionPane.showMessageDialog(null,"USER or DISABLE
PASSWORD!","",JOptionPane.ERROR_MESSAGE);
            txtPass.setText("");
            txtId.setText("");

        }
    }


    private void exitForm(java.awt.event.WindowEvent evt) {
        System.exit(0);
    }

    public static void main(String args[]) {
        new Login().show();
    }

}
```

APÊNDICE E – Classe ConnectionDatabase

```
/**
 *
 * @author  vasco
 */

import java.sql.*;

public class ConnectionDatabase {

    private Connection connection;
    private String password;
    private String userName;
    private String name;
    private Statement stm;


      public Connection connect(){
        Connection c;
        try{
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            c = DriverManager.getConnection("jdbc:odbc:Autenticacao");
            System.out.println("Conectado");
        }catch (SQLException sqlEx){
            System.out.println("SQL Exception");
            sqlEx.getStackTrace();
            c = null;
        }
        catch (ClassNotFoundException classNotFound){
            System.out.println("Class not Found");
            classNotFound.getStackTrace();
            c = null;
        }
        return c;
    }


    public boolean consulta(String user,String password){
        boolean ok = false;
        connection = connect();
        try{
            stm =  connection.createStatement();
            ResultSet rs;
            rs = stm.executeQuery("Select * from LOGIN where USER =
'"+user+"';");
            if (!rs.next()){
                System.out.println("Usuário não Existe");
            }
            ResultSetMetaData rsmd = rs.getMetaData();
            if (password.equals(rs.getString(2))){
                ok = true;
            }
            else{
                ok = false;
                System.out.println("Senha Incompativel");
            }
        }catch (SQLException sqlException){
            sqlException.printStackTrace();
        }
```

```
        try{
            connection.close();
        }catch (SQLException sqlException){
            sqlException.printStackTrace();
        }
        return ok;
    }


      public boolean searchUser(String nUser, String nPassword) {
        String consulta =
        "SELECT * FROM LOGIN";
        ResultSet rs;
        try {
            rs = stm.executeQuery(consulta);


            while (!rs.isLast()) {
                rs.next();

                userName = rs.getString("user");
                password  = rs.getString("password");

                if ((userName.equals(nUser)) &&
(password.equals(nPassword)))
                    return true;
            }

            return false;
        }

        catch (SQLException e) {
            e.getMessage();
        }
        return false;

    }


      public void ToCloseBD()
    {
        try {   connection.close(); }

        catch ( SQLException e ) {
            System.err.println( "It was not possible to close the
connection" + e );
            System.exit( 1 );
        }
    }


    public void setName(String Nname) {
        name = Nname;    }


      public void setPasswordl(String NPassword) {
        password = NPassword;    }


      public void setUserName(String NuserName) {
```

```
        userName = NuserName;    }


      public void ToSaveUser() {
        String sql;
        sql = "INSERT INTO LOGIN(USER,PASSWORD,NAME) VALUES ( '" +
        userName+"','"+password+"','"+name+"' )";
        int linha = 0;
        try {

            linha = stm.executeUpdate(sql);
        }
        catch (SQLException e ) {
            e.getMessage();
        }

        if (linha > 0) {
            System.out.println("Usuário salvo");
        }
    }
}
```

APÊNDICE F – Classe Register

```
/**
 *
 * @author  vasco
 */
public class Register extends javax.swing.JFrame {

      private JButton jButton1;
    private JButton jButton2;
    private JLabel jLabel1;
    private JLabel jLabel2;
    private JLabel jLabel3;
    private JLabel jLabel4;
    private JLabel jLabel5;
    private JPasswordField jPasswordField1;
    private JPasswordField jPasswordField2;
    private JTextField jTextField1;
    private JTextField jTextField2;
      private  ConnectionDatabase bc = new ConnectionDatabase();


    public Register() {
        initComponents();
    }


    private void initComponents() {
        jLabel1 = new JLabel();
        jLabel2 = new JLabel();
        jTextField1 = new JTextField();
        jLabel3 = new JLabel();
        jTextField2 = new JTextField();
        jLabel4 = new JLabel();
        jLabel5 = new JLabel();
        jPasswordField2 = new JPasswordField();
        jButton1 = new JButton();
        jButton2 = new JButton();
        jPasswordField1 = new JPasswordField();


        getContentPane().setLayout(null);

        setTitle("Registrar novo usuário");
        addWindowListener(new java.awt.event.WindowAdapter() {
            public void windowClosing(java.awt.event.WindowEvent evt) {
                exitForm(evt);
            }
        });

        jLabel1.setFont(new java.awt.Font("Arial", 1, 18));
        jLabel1.setText("Registrar novo usu\u00e1rio");
        getContentPane().add(jLabel1);
        jLabel1.setBounds(100, 10, 220, 22);

        jLabel2.setFont(new java.awt.Font("MS Sans Serif", 1, 11));
        jLabel2.setText("Nome:");
        getContentPane().add(jLabel2);
        jLabel2.setBounds(10, 50, 50, 15);
```

```
        getContentPane().add(jTextField1);
        jTextField1.setBounds(150, 50, 280, 20);

        jLabel3.setFont(new java.awt.Font("MS Sans Serif", 1, 11));
        jLabel3.setText("Nick:");
        getContentPane().add(jLabel3);
        jLabel3.setBounds(10, 80, 90, 15);

        getContentPane().add(jTextField2);
        jTextField2.setBounds(150, 80, 160, 20);

        jLabel4.setFont(new java.awt.Font("MS Sans Serif", 1, 11));
        jLabel4.setText("Senha:");
        getContentPane().add(jLabel4);
        jLabel4.setBounds(10, 110, 90, 15);

        jLabel5.setFont(new java.awt.Font("MS Sans Serif", 1, 11));
        jLabel5.setText("Confirmar Senha:");
        getContentPane().add(jLabel5);
        jLabel5.setBounds(10, 140, 130, 15);

        getContentPane().add(jPasswordField2);
        jPasswordField2.setBounds(150, 140, 140, 22);

        jButton1.setText("Salvar");
        jButton1.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                jButton1ActionPerformed(evt);
            }
        });

        getContentPane().add(jButton1);
        jButton1.setBounds(220, 180, 90, 23);

        jButton2.setText("Cancelar");
        jButton2.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                jButton2ActionPerformed(evt);
            }
        });

        getContentPane().add(jButton2);
        jButton2.setBounds(110, 180, 90, 23);

        getContentPane().add(jPasswordField1);
        jPasswordField1.setBounds(150, 110, 140, 22);

        java.awt.Dimension screenSize =
java.awt.Toolkit.getDefaultToolkit().getScreenSize();
        setBounds((screenSize.width-448)/2, (screenSize.height-250)/2, 448,
250);
    }

    private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
        this.dispose();
    }

    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
        if ( verifyRegister() ) {
            insert();
```

```
            clear();
        }
        else {
            javax.swing.JOptionPane.showMessageDialog(null,
            "Erro ao tentar inserir\nPreencha corretamente todos os
campos","Erro",
            javax.swing.JOptionPane.ERROR_MESSAGE);
            clear();
        }
    }

    private void exitForm(java.awt.event.WindowEvent evt) {
        System.exit(0);
    }

    private void insert() {

        bc.setName(jTextField1.getText() );
        bc.setPasswordl(jPasswordField1.getText() );
        bc.setUserName(jTextField2.getText() );

        bc.ToSaveUser();
        System.out.println("PAssou aqui...");
        javax.swing.JOptionPane.showMessageDialog(null, "Aluno inserido com
sucesso",
        "Inserção",javax.swing.JOptionPane.INFORMATION_MESSAGE);
    }

    private void clear() {
        jTextField1.setText("");
        jTextField2.setText("");
        jPasswordField1.setText("");
        jPasswordField2.setText("");
    }

    private boolean verifyRegister() {
        System.out.println("Verificando o Cadastro");
        if ( jTextField1.equals("") ) {
            System.out.println("O campo NOME está em branco");
             return false;
        }

        else if ( jTextField2.equals("") ) {
            System.out.println("O campo User Name está em branco");
            return false;
        }

        else if ( jPasswordField1.equals("") ) {
            System.out.println("O campo senha está em branco");
            return false;
        }

        else if ( jPasswordField2.equals("") ) {
            System.out.println("O campo SENHA Confirmação está em branco");
            return false;
        }

        else if
(!jPasswordField1.getText().equals(jPasswordField2.getText() )) {
            System.out.println("SENHA INVÁLIDA VERIFIQUE");
```

```
            return false;
        }

        else if ( bc.consulta(jTextField2.getText(),
jPasswordField1.getText() ) ) {
            System.out.println("ALUNO JÁ CADASTRADO");
            return false;
        }

        else return true;
    }

    public static void main(String args[]) {
        new Register().show();
    }

}
```

APÊNDICE G – Classe CreateImage

```
/**
 *
 * @author  vasco
 */


import java.awt.*;
import java.awt.image.*;
import java.util.*;
import java.io.*;
import javax.imageio.ImageIO;

public class CreateImage extends BufferedImage {

       public CreateImage(int w, int h) {
        //super classe, largura, altura
        super(w,h, TYPE_BYTE_GRAY);
    }


    static public CreateImage createImage(String filename) throws
IOException {
        Image im =  ImageIO.read(new File(filename));

        BufferedImage bi = (BufferedImage) im;
       CreateImage gi = new CreateImage(bi.getWidth(), bi.getHeight());
        gi.setData(bi.getRaster());
        return gi;
    }


    static public void main(String[] args) throws Exception {
        CreateImage gi = createImage(args[0]);
    }

}
```

APÊNDICE H – Classe MyCanvas

```
/**
 *
 * @author  vasco
 */



class MyCanvas extends Canvas {

      BufferedImage bi;


    public MyCanvas(BufferedImage b) {
        bi = b;
    }


    public void paint(Graphics g) {
        if ( getImage() != null )
            g.drawImage(getImage(), 0, 0, null);
    }


    public BufferedImage getImage() {
        return bi;
    }


    public void setImage(BufferedImage b) {
        bi = b;
        setSize(new Dimension(b.getWidth(),b.getHeight()));
    }

}
```