UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

PROPAGAÇÃO DA INTERFACE DO VXT USANDO O MODELO CLIENTE/SERVIDOR

ELTON FERNANDO GOEDERT

BLUMENAU
2006

2006/2-10

ELTON FERNANDO GOEDERT

PROPAGACÃO DA INTERFACE DO VXT USANDO O MODELO CLIENTE/SERVIDOR

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciências da Computação — Bacharelado.

Prof. Mauro Marcelo Mattos, Dr. - Orientador

BLUMENAU
2006

2006/2-10

PROPAGAÇÃO DA INTERFACE DO VXT USANDO O MODELO CLIENTE/SERVIDOR

Por

ELTON FERNANDO GOEDERT

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

	_____
Presidente:	Prof. Mauro Marcelo Mattos, Dr. – Orientador, FURB
	_____
Membro:	Prof. Antonio Carlos Tavares, Especialista – FURB
	_____
Membro:	Prof. Miguel Alexandre Wisintainer, Mestre – FURB

Blumenau, 13 de dezembro de 2006

Dedico este trabalho a minha família, aos meus amigos e aos mestres desta universidade, especialmente aqueles que contribuíram com a realização deste.

AGRADECIMENTOS

A Deus, pela oportunidade de concluir mais uma etapa na minha vida.

Aos meus pais e ao meu irmão, por sempre me incentivarem e me apoiarem na conquista deste objetivo.

Aos meus amigos, por estarem comigo sempre nas horas que precisei, compreendendo a minha ausência durante esta realização.

Ao meu orientador, Mauro M. Mattos, por ter me incentivado e motivado para a realização deste trabalho.

Todas as grandes descobertas e invenções foram sonhos no início. O que se pressente hoje se realiza amanhã.

Hellmuth Unger

RESUMO

Este trabalho apresenta uma solução incorporada ao projeto VXt, para viabilizar a distribuição da interface com o usuário do software, para uma sala de aula. O projeto utiliza comunicação cliente/servidor e envolve o uso de concorrência com threads para uma distribuição eficaz e segura dos dados.

Palavras-chave: VXt. Middleware. Modelo cliente/servidor.

ABSTRACT

This work describes a software tool that was included to VXt project in order to provide it with a capability to broadcast its user interface. The project is based in a client/server model and employ threads to build a secure broadcast service.

Keywords: VXt. Middleware. Client/server model.

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

LISTA DE SIGLAS

API – *Application Programming Interface*

CGA – *Color Graphics Adapter*

CPU – *Central Processing Unit*

C/S – Cliente/Servidor

CVS – *Concurrent Versions System*

EISA – *Extended Industry Standard Architecture*

J2SDK – Java *2* Software Development Kit

I/O – *Input/Output*

INTR – *Interrupt Request*

IP – Internet *Protocol*

IRET – *Interrupt Return*

ISA – *Industry Standard Architecture*

MDA – *Monocromatic Display Adapter*

MS-DOS – Microsoft *Disk Operating System*

MVC – *Model-View-Controller*

PCI – *Peripheral Component Interconnect*

PIC – *Programmable Interrupt Controller*

RF – Requisito Funcional

RNF – Requisito Não Funcional

TCC – Trabalho de Conclusão de Curso

TCP – *Transmission Control Protocol*

VXt – Virtual XT

SUMÁRIO

1 INTRODUÇÃO

Conforme Gagné, Brigss e Wagner (1992 apud CASAS, 1999), instrução, no contexto da escola, é um processo oferecido por instrutores humanos e envolve o professor e o aprendiz. Os méritos da instrução proporcionada por professores humanos são os aspectos dinâmicos e interativos do ensino.

> Aprender por projetos é uma forma inovadora de romper com as tradições educacionais, dando um formato mais ágil e participativo ao trabalho de professores e educadores. Trata-se mais do que uma estratégia fundamental de aprendizagem, sendo um modo de ver o ser humano construir, aprendendo pela experimentação ativa do mundo. Ao elaborar seus projetos, o professor conduzirá seus alunos a um conjunto de interrogações, quer sobre si mesmos, quer sobre o mundo à sua volta, levando o aluno a interagir com o desconhecido ou com novas situações, buscando soluções para os problemas (FAGUNDES, 1998 apud VEIGA, 2001).

O currículo do curso de Ciências da Computação da Universidade Regional de Blumenau (FURB) estabelece como obrigatórias as disciplinas Arquitetura de Computadores e Sistemas Operacionais. A disciplina Arquitetura de Computadores apresenta conceitos básicos sobre os componentes de hardware de um computador. O estudo de sistemas operacionais pressupõe o conhecimento de conceitos básicos de arquitetura de computadores. No curso de Sistemas de Informação da FURB, o currículo também estabelece como obrigatórias as disciplinas Arquitetura de Computadores I e Sistemas Operacionais.

Em uma análise realizada sobre trabalhos acadêmicos desenvolvidos na disciplina de Sistemas Operacionais, os professores Mauro Marcelo Mattos e Antonio Carlos Tavares observaram que havia uma deficiência no entendimento de como realmente as ações eram executadas por um processador. Em função disto, decidiram iniciar o desenvolvimento de uma ferramenta acadêmica que facilitasse o processo de compreensão sobre o funcionamento de um processador. Esta ferramenta denominou-se VXt, fazendo uma alusão ao nome do computador que na época utilizava o processador simulado – um *Personal Computer* (PC) XT (MATTOS; TAVARES; OLIVEIRA, 1998).

A proposta original do projeto iniciado em 1997 consistia na implementação em software de uma *Central Processing Unit* (CPU) Intel 8086, para permitir a partir dela, a elaboração de testes sobre o processador, visando a demonstração de conceitos básicos de sistemas operacionais e arquitetura de computadores (MATTOS; TAVARES; OLIVEIRA, 1998).

Várias versões foram construídas e atualmente o VXt encontra-se na versão 5.01. Uma das limitações desta versão é que o software só executa em ambiente Windows. Esta versão

implementa uma arquitetura cliente/servidor primitiva que além de problemas de robustez, dificulta em muito a administração de entrada/saída de alunos após o início de uma sessão de estudo.

Segundo Bochenski (1995, p. 1), a computação cliente/servidor é a mudança mais importante ocorrida na tecnologia de informática em toda a história.

> Embora os primeiros aplicativos cliente/servidor se enquadrassem apenas em determinadas categorias – como acesso a dados e sistemas de apoio a decisão –, a medida que as tecnologias relacionadas vão se aperfeiçoando, a variedade de aplicativos implementada como sistemas de cliente/servidor continua a se expandir. (BOCHENSKI, 1995, p. 1).

Diante do exposto, o presente trabalho visa estender o software VXt para permitir a propagação da interface com o usuário do mesmo entre os alunos de uma sala de aula. O modelo descrito neste projeto contempla dois módulos: (i) um módulo de interface entre o VXt (escrito em Delphi) e um *middleware* escrito em Java; (ii) um módulo de interface com o usuário escrito totalmente em Java.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é ampliar o software VXt incorporando uma facilidade de propagação da interface do usuário entre diversos clientes em uma sala de aula.

Os objetivos específicos do trabalho são:

a) implementar um *middleware* de comunicação entre o simulador da CPU e as interfaces dos clientes;
b) permitir ao usuário do simulador do processador que a interface possa ser executada em uma máquina diferente daquela onde o simulador está sendo rodado.

1.2 ESTRUTURA DO TRABALHO

O texto está estruturado em 4 capítulos. No segundo capítulo, é apresentada a fundamentação teórica utilizada para o desenvolvimento do trabalho, que trata do estudo do VXt, do modelo cliente/servidor, do Softx86 e do MVC. No terceiro capítulo é apresentado o

desenvolvimento, incluindo a especificação dos requisitos e do caso de uso, a modelagem estrutural das classes, as ferramentas utilizadas no processo, a implementação e a operacionalidade do VXt. Por último, no capítulo quatro, são apresentadas as conclusões e sugestões de trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

A fundamentação teórica aborda os aspectos teóricos relacionados à contextualização do trabalho. Engloba uma apresentação do projeto VXt em suas várias versões apresentadas em (MATTOS et. al., (2004); MATTOS; TAVARES, (1999a); MATTOS; TAVARES, (1999b); MATTOS; OLIVEIRA, (1999); MATTOS; TAVARES, (1998); MATTOS; TAVARES, (1997)). Comenta a utilização do modelo cliente/servidor para a transmissão dos dados. Demonstra a integração entre o VXt e a biblioteca Softx86. Apresenta ainda as principais características de um trabalho que utilizou o VXt como substrato para o desenvolvimento de um tutorial de *assembl*y. Trata também do padrão MVC como uma forma de viabilizar a separação entre a interface e o modelo viabilizada por um *middleware* de comunicação. E por último relata as principais características dos trabalhos correlatos que constituem a base para a criação da ferramenta desenvolvida.

2.1 O PROJETO VXT

O VXt é um programa que começou a ser desenvolvido em 1997, e tem como objetivo demonstrar a simulação de um processador Intel 8086.

Conforme Mattos (2003, p. 50), “embora haja varias linhas arquitetônicas, via de regra o funcionamento de um processador pode, abstratamente, ser descrito como apresentado em Tanenbaum (1984, p. 22) através da implementação em linguagem pascal” (Figura 1).

Os elementos necessários para a construção de um simulador de um determinado processador são:

a) um *array* de memória (Figura 1 - linha 3);
b) um apontador de instruções que indica o endereço de memória que contém uma instrução a ser decodificada e executada (Figura 1 - linha 7);
c) um *flag* indicando se o processador deve executar um próximo laço de execução (Figura 1- linha 9);
d) um registrador de instruções que armazena a instrução a ser decodificada e executada (Figura 1 - linha 10);
e) um laço que envolve as operações de busca de uma instrução na memória (ciclo de

fetch) e execução da mesma (ciclo de *execute*) até que uma instrução de parada (*halt*) seja encontrada ou um erro ocorra (instrução não identificada).

```
1    Type Word = ....;                  (* especificação do tipo word *)
2     Ender = ....;                     (* endereço*)
3     Mem = array[0..4095] of Word; (* alocação  da memória do "computador simulado" *)
4     (* ..... *)
5    procedure interpretador (memoria:Mem;EndInicio:Ender);
6    var
7     ProgramCounter,                   (* apontador da proxima instrução a ser executada *)
8     DataLocation      : Ender;        (* apontador para endereço dos operandos da instrução *)
9     RunBit            : 0..1;         (* flag indicando quando parar o processador *)
10    InstrType         : integer;      (* tipo da instrução a ser executada *)
11   begin
12    ProgramCounter := EndInicio;(*endereço da primeira instrução a ser executada quando a máquina é ligada *)
13    RunBit := 1;                      (* estabelece o padrão de repetição do processador *)
14    While RunBit = 1 do
15    begin
16      (* busca a próxima instrução e armazena no registrador de instruções do processador – FETCH *)
17      InstrRegister := memoria[ProgramCounter];
18      (* avança o ProgramCounter para apontar para a próxima instrução a ser executada *)
19      ProgramCounter := ProgramCounter + 1;
20      (* decodifica a instrução e classifica o seu tipo (aritmética,lógica, ...)  *)
21      DeterminaTipoInstrucao(InstrRegister,TipoInstr);      (* "chamada de procedimento" *)
22      (* identifica e localiza o endereço dos operandos da instrução- se houverem *)
23      IdentificaOperandos (TipoInstr,InstrRegister,DataLocation,NecessidaDados);
24      (* busca dados dos operandos na memória – se houverem *)
25      if NecessitaDados then  Operandos := memoria[ DataLocation];
26      (* avança o processo executando a instrução *)
27      Execute (TipoInstr,Operandos,memoria, ProgramCounter, RunBit);
28    end; (* while *)
29   end; (* Interpretador *)
```

Fonte: Mattos (2003, p. 57).

Figura 1 - Pseudocódigo que implementa um interpretador de máquina virtual

Portanto, a concepção lógica do projeto VXt está baseada no modelo apresentado na (Figura 1). A evolução do projeto vem sendo relatada em uma série de publicações, as quais são apresentadas nesta seção, permitindo um resgate histórico do projeto.

2.1.1 A primeira publicação

A primeira publicação sobre o projeto VXt (MATTOS; TAVARES, 1997, p. 70) apresentava como motivação, a caracterização de problemas de aprendizagem na disciplina de Sistemas Operacionais.

> Analisando-se o resultado dos trabalhos realizados na disciplina de Sistemas Operacionais, observou-se que havia uma deficiência no entendimento de como realmente as ações eram executadas por um processador, e quais as implicações disto em termos de arquitetura de um sistema operacional. A ausência de uma ferramenta didática que permita exemplificar os conceitos de modo mais claro, estimulou o desenvolvimento do VXt. (MATTOS; TAVARES, 1997, p. 70).

"Em função disto, iniciou-se um projeto em sala de aula, com o intuito de complementar os conhecimentos básicos sobre arquitetura de computadores, o qual culminou com a implementação em software do processador Intel 8086." (MATTOS; TAVARES, 1997, p. 71).

Para a escolha do processador foram considerados os seguintes aspectos (MATTOS; TAVARES, 1997, p. 71):

a) o processador a ser implementado deveria ser de conhecimento dos alunos, tendo em vista evitar um esforço a mais para conhecer um novo conjunto de instruções;
b) o processador alvo, deveria possuir todo um aparato de ferramentas de desenvolvimento tais como, compiladores, montadores, depuradores e ambientes de desenvolvimento adequados, para evitar o esforço desnecessário na construção de tais ferramentas;
c) o processador alvo deveria executar um sistema operacional que fosse de conhecimento do público-alvo, tendo em vista evitar o esforço desnecessário (em função dos objetivos do trabalho) no aprendizado da utilização do mesmo;
d) deve haver disponibilidade de literatura tendo em vista permitir a implementação do conjunto de instruções do processador.

A partir destes requisitos, selecionou-se o processador Intel 8086, tendo em vista que (MATTOS; TAVARES, 1997, p. 71):

a) ele atende ao requisito (a) acima descrito na medida em que os alunos da disciplina de arquitetura de computadores desenvolvem pequenas aplicações utilizando a linguagem *assembly* deste processador;
b) ele atende ao requisito (b), na medida em que há toda uma estrutura que suporta o desenvolvimento de aplicações para a plataforma Intel;
c) ele atende ao requisito (c), na medida em que o sistema operacional MS-DOS, já é de conhecimento dos alunos, não necessitando um esforço de treinamento adicional para esta plataforma;
d) a biblioteca da Universidade dispõe de literatura suficiente para permitir aos alunos o desenvolvimento da referida aplicação.

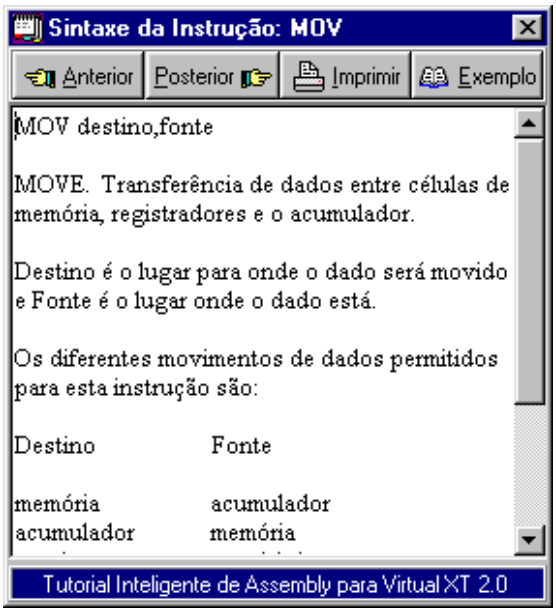
2.1.2 O primeiro TCC desenvolvido sobre o VXt

O TCC desenvolvido por Linzmeier (1999), utilizou como plataforma de trabalho o VXt versão 2.0 (Figura 2).

Fonte: Linzmeier (1999, p. 39).

Figura 2 – Interface do VXt 2.0

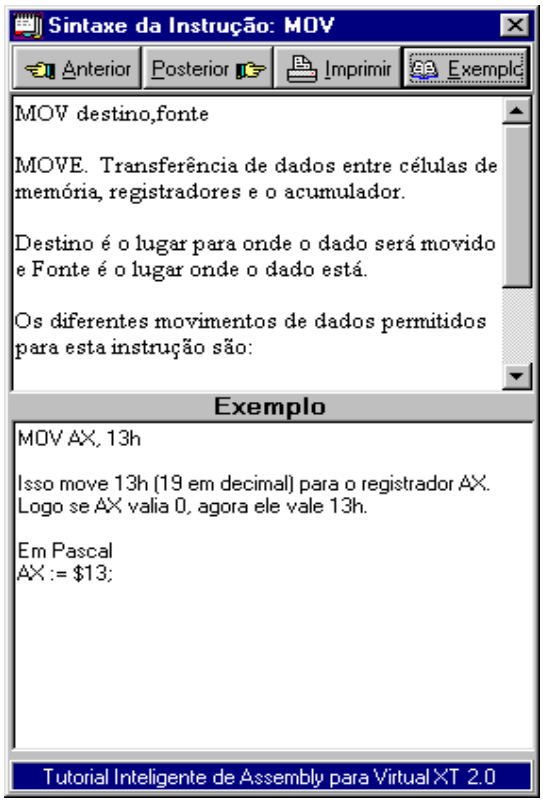
O protótipo consiste basicamente em exibir informações sobre cada instrução da linguagem *assembly* (Figura 3), apresentadas uma a uma na tela do VXt, no campo destinado a apresentação do *opcode* (código que identifica uma instrução de hardware do processador). da instrução sendo executada.



Fonte: Linzmeier (1999, p. 40).

Figura 3 – Interface do tutorial

Pressionando-se o botão ‘exemplo’, a janela do tutorial será expandida, mostrando o exemplo, conforme apresentado na Figura 4.



Fonte: Linzmeier (1999, p. 39).

Figura 4 – Janela do tutorial com exemplo

Para explicações mais detalhadas sobre *assembly* há um botão de *help* que pode ser ativado a partir da janela principal do VXt.

2.1.2.1 Funcionamento do Protótipo

O VXt 2.0 funciona da seguinte forma: após ser aberto um arquivo executável (.exe ou .com), apertando-se no botão *play* ele é executado, passando para a próxima instrução e assim sucessivamente. Para ativar o tutorial é só passar com o mouse em cima do campo que identifica a instrução, automaticamente a janela do tutorial se abrirá mostrando a sintaxe da instrução. Para ver um exemplo de uso é só clicar no botão exemplo da janela da sintaxe. Há também a opção para visualizar a impressão da instrução atualmente apresentada.

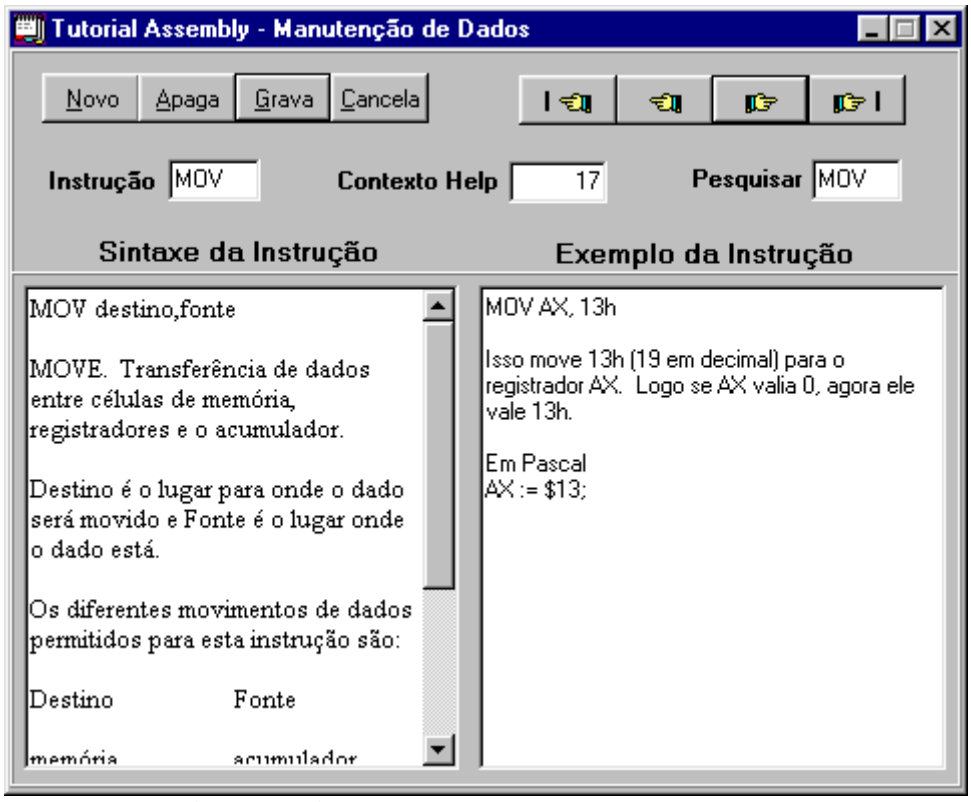
Caso não se queira fazer o acompanhamento da execução das instruções pelo tutorial, basta desativá-lo clicando no último botão da barra de ferramentas do Vxt 2.0 (Figura 5).

Fonte: Linzmeier (1999, p. 40).
Figura 5 – Ícone para ativar/desativar o tutorial

2.1.2.2 Construção do Protótipo

As instruções e os exemplos que compõem o tutorial são gravadas numa tabela chamada hints.db do tipo Paradox. Para fazer o gerenciamento da tabela foi desenvolvido um programa em Delphi 3.0, chamado de *Manutenção de Dados* – como mostra a Figura 6, que permite pesquisar uma instrução a fim de consultá-la, excluí-la ou mesmo alterá-la, caso a instrução desejada não for encontrada, será mostrado a instrução mais próxima àquela fornecida como argumento.

Desejando incluir uma nova instrução pressiona-se o botão 'novo' para cadastrá-la, todas as modificações são confirmadas no botão 'grava' ou canceladas no botão 'cancela'. O botão 'apaga' exclui a instrução da tabela.



Fonte: Linzmeier(1999, p. 38).
Figura 6 – Manutenção de dados

O tutorial foi baseado no funcionamento do *hint* do Delphi, que é uma propriedade que permite apresentar uma breve descrição a um determinado objeto. O mecanismo do tutorial apresentará o texto, quando se posicionar o mouse sobre o campo que contém a instrução *assembly*, abrindo assim a janela do tutorial, pesquisando a instrução mostrada na tela no banco de dados, apresentando o conteúdo do registro encontrado.

O Quadro 1, descreve o algoritmo utilizado.

```
Procedimento que ativa a Hint;
Variáveis locais;
Inicio
     Guarda a instrução mostrada no Vxt;
     Guarda caracter até encontrar uma espaço ou barra;
     Se for barra então
         Copia para variável até a barra
     Senão se for  espaço então
         Copia para variável até o espaço
     Senão
         Copia 5 primeiros caracteres;
    Fim;
    Pesquisa o conteúdo da variável na tabela Hints.db;
    Mostra conteúdo pesquisado na tabela;
Fim;
```

Fonte: Linzmeier (1999, p. 37).

Quadro 1 - Procedimento para capturar a instrução do Vxt

2.1.3 Descrição da arquitetura do VXt versão 2.5

O projeto VXt continuou evoluindo e, em 1998, foi publicado um trabalho (MATTOS; TAVARES; OLIVEIRA, 1998, p. 138) descrevendo como haviam sido implementadas as principais estruturas do VXt. Esta seção apresenta um resumo daquela publicação.

2.1.3.1 A arquitetura de memória

A arquitetura de memória do 8088/86 utiliza o modelo segmentado. Como um processador de 16-bits com registradores de 16 bits é capaz de endereçar somente 64Kb usando mecanismos de endereçamento direto, o modelo segmentado foi projetado para permitir o acesso a 1 MB de memória. Cada posição física da memória é endereçada por 2 valores de 16 bits - um segmento e um deslocamento (*segment*:*offset*).

O valor de segmento determina o início de uma região de 64Kb, e o valor de deslocamento (*offset*) aponta para o byte sendo endereçado. As posições de memória são

descritas através de endereços lógicos expressos no formato segmento:deslocamento (ou *segment*:*offset*). Valores de segmento são carregados para um dos 4 registradores de segmento que apontam para o início de 4 segmentos de memória endereçáveis a cada momento. Quando uma posição de memória é acessada, o valor do registrador de segmento é multiplicado por 16 e adicionado ao valor do deslocamento para obter-se o endereço desejado.

Em função da operação de multiplicação, os endereços de segmento são múltiplos de 16 bytes. Como os endereços são calculados aritmeticamente, é possível que o endereço de início de um segmento aponte para uma região já ocupada por outro segmento. Cabe portanto ao programa de aplicação controlar esta situação.

Nas versões anteriores, a memória do VXt era declarada como mostrado no Quadro 2, ou seja, uma estrutura estática e global limitada a um tamanho máximo de aproximadamente 60000 bytes. Esta restrição havia enquanto o projeto utilizava como ambiente de desenvolvimento o pacote Turbo Pascal 5.0. Quando o projeto migrou para o ambiente Delphi, esta restrição deixou de existir. Hoje o VXt possui um *array* de memória de 1Mb.

```
const
      MaxTamMemoriaReal = 48000;
type
   tipo_memoria =
array[0..MaxTamMemoriaReal] of byte;
var
```

Fonte: Mattos, Tavares e Oliveira (1998, p. 140).

Quadro 2 – Declaração estática da memória

Verificou-se então que, a estrutura de memória declarada como mostrado anteriormente, descaracterizava a implementação real na medida em que, um acesso como por exemplo: `AX := memória[segmento:deslocamento]` esconderia o fato de que para um dado sair efetivamente da memória e ser transferido para um registrador qualquer da CPU, ele obrigatoriamente teria que passar pelo barramento de dados (o mesmo aplica-se ao barramento de endereços).

Em função disto, a versão atual encapsulou a estrutura de dados que implementa a memória do VXt, disponibilizando somente 2 operações para acesso a mesma conforme apresentado no Quadro 3. Dessa forma é possível, numa futura versão, o estudo e implementação das diversas arquiteturas de barramento disponíveis (ISA, EISA, Micro Canal, PCI, etc.) sem haver a necessidade de alteração completa do código-fonte.

```
{armazena o valor passado como parametro no endereco segmento:offset}
procedure REMSegOfs(segmento,offset:word;valor:byte);
{Retorna o conteudo da posicao de memoria enderecada por
segmento:offset}
function RDMSegOfs(segmento,offset:word):byte;
```

Fonte: Mattos, Tavares e Oliveira (1998, p. 140).

Quadro 3 - Interface para acesso a memória do VXt

2.1.3.2 O conjunto de registradores

O conjunto de registradores é composto de registradores gerais, especiais e de segmentos, conforme apresentado na Quadro 4. Os registradores AX, BX, CX, DX, SI e DI são usados para armazenar operandos de operações lógicas e aritméticas. Estes registradores podem ser usados nas mais simples instruções, e cada um tem algumas funções no conjunto de instruções mais complexo.

Registradores de segmento	**Registradores Gerais**		**Registradores Especiais**
CS = Code Segment	**AX:** acumulador	**BX:** base *	**IP:** instruction pointer
SS - Stack Segment	**CX:** contador	**DX:** dados, I/O	**FLAGS:** flags de status
DS - Data Segment	**BP:** base pointer *	**SP:** stack pointer	
ES - Extra Segment	**SI:** source index *	**DI:**destination index*	
* podem ser usados como registradores de deslocamento (offset)			

Fonte: Mattos, Tavares e Oliveira (1998, p. 141).

Quadro 4 - Conjunto de Registradores da CPU Intel 8086

O registrador AX é usado como acumulador *default* (padrão) na maioria das instruções. O registrador BX é usado como registrador de endereçamento base; o registrador CX é usado como um contador em operações de *loop*; e o registrador DX é usado em operações de I/O.

Os registradores SI e DI podem ser usados como deslocamentos origem/destino em instruções especiais para manipulação de *strings* para realizar transferência de/para posições de memória. Os registradores BX, SI, DI, e BP são os únicos registradores que podem ser usados como registradores de índice ou *offset* em operações de cálculo de endereço.

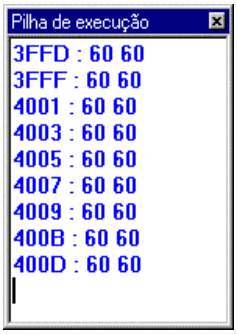
Como afirmado anteriormente, os registradores de segmento estabelecem 4 segmentos de 64Kb. Cada registrador de segmento possui uma finalidade especial. O registrador CS determina o endereço base do segmento que contém a seqüência de instruções a serem executadas - é chamado de registrador de código (*code segment*). O 8088/86 busca todas as

instruções a serem executadas a partir deste segmento de código usando como *offset* o conteúdo do registrador IP. A combinação CS:IP forma o *contador de instruções* (ou apontador de instruções - *instruction pointer*). Seu conteúdo somente pode ser alterado a partir da execução de instruções de transferência de controle de fluxo tais como : c*all*, *jmp*, interrupções e exceções.

Quando uma transferência ocorre sem o registrador CS mudar, esta transferência é chamada *near* (próxima), porque ela refere-se a um endereço dentro do segmento atualmente apontado pelo registrador CS. Quando a transferência ocorre para um endereço fora do segmento atualmente apontado por CS, o valor de CS tem que ser alterado e a transferência é dita *far* (longe).

O 8088/86 usa uma pilha (*stack*) para facilitar a passagem de parâmetros e a criação de *registros de ativações locais* (variáveis locais). O registrador SS sempre contém o endereço base do início da área de pilha, e o registrador SP sempre aponta para o topo desta pilha. A pilha é referenciada implicitamente nas operações de *push, pop, call* e outras operações de controle de transferência de fluxo. Ao contrário do registrador CS, o registrador SS pode ter seu valor alterado explicitamente através de uma atribuição, permitindo desta forma que os programadores definam pilhas dinamicamente.

A Figura 7 apresenta a janela que permite a visualização da pilha de execução no VXt.



Fonte: Mattos, Tavares e Oliveira (1998, p. 141).

Figura 7 - Pilha de execução

O registrador BP é geralmente usado como um apontador que demarca o início das variáveis locais a um procedimento, também chamado de *stack frame*. Quando o registrador BP é usado como registrador índice no cálculo de um endereço, o registrador SS é usado como registrador de segmento por *default*.

Os registradores DS e ES permitem a especificação de segmentos de dados. Geralmente o registrador DS é usado para referenciar ao segmento de dados *default* de uma aplicação, e o registrador ES é usado para outras referências fora do escopo do segmento de

dados *default*. A maioria das instruções que referenciam memória usam o registrador DS por padrão.

O Quadro 5 apresenta a declaração das estruturas de dados que implementam os registradores do VXt. Cabe destacar a estratégia adotada para permitir a manipulação dos registradores AX, BX, CX e DX (todos de 16 bits) como se fossem registradores de 8 bits através da declaração *absolute*. Dessa forma, é possível movimentar-se um dado de 16 bits para o registrador AX, por exemplo, e obter somente os primeiros 8 bits (partindo-se da esquerda para a direita) ou a chamada parte baixa do valor e os restantes 8 bits ou chamada parte alta do valor, independentemente e sem a necessidade de processamento desnecessário.

```
type
 LowHigh = record L:byte; H:byte; end;
var
 CS,DS,ES,SS,SP,BP,SI,DI,IP,flags: word;
 A,B,C,D: LowHigh;

 AX: word absolute A;
 BX: word absolute B;
 CX: word absolute C;
 DX: word absolute D;
```

Fonte: Mattos, Tavares e Oliveira (1998, p. 142).

Quadro 5 - Declaração dos registradores do VXt

O registrador *flags* contém as indicações de status ou códigos de condição. Estas indicações permitem o resultado de uma instrução influenciar nas instruções seguintes, preservando o estado de uma operação lógica ou aritmética. Este registrador possui 16 bits, sendo que somente 9 são usados (Quadro 6).

Carry flag (CF)	Indica se ocorreu um "vai um" após uma instrução aritmética ter sido executada
Parity flag(PF)	indica se o número de bits da operação é par ou impar
Auxiliar carry flag (AF)	indica condições especiais de "vai um"';
Zero flag(ZF)	indica que o resultado da operação resultou no valor zero
Sign flag (SF)	indica se o resultado da operação é negativo;
Trap flag(TF)	indica situação de processamento "single step"
Interrupt flag(IF)	indica se é possível o processador ser interrompido ou não
Direction flag(DF)	indica a direção de transferência dos dados
Overflow flag(OF)	Indica se a função aritmética executada estourou (overflow) a capacidade de armazenamento do registrador.

Fonte: Mattos, Tavares e Oliveira (1998, p. 142).

Quadro 6 - Relação dos bits do registrador de *flags* da CPU Intel 8086

O Quadro 7 apresenta a declaração das constantes utilizadas para identificar e manipular individualmente cada um dos bits de *flags* no VXt.

```
const
  fcf=0; fpf=2; faf=4; fzf=6;  fsf=7; ftf=8; fif=9; fdf=10; fof=11;
```

Fonte: Mattos, Tavares e Oliveira (1998, p. 143).

Quadro 7 – Declaração dos identificadores dos bits do registrador de *flags*

2.1.3.3 O mecanismo de busca e execução de instruções

A filosofia de funcionamento de um processador é um processo relativamente simples e, conforme apresentado na seção 2.1, consiste basicamente de duas fases: uma fase de busca da próxima instrução a ser executada e, uma fase de execução da mesma. A estas fases, dá-se o nome de ciclo de *fetch* e *execute*, respectivamente.

No VXt, este mecanismo pode ser identificado através do Quadro 8. O acesso a memória é realizado através da chamada ao procedimento `RDMSegOfs (CS, IP)` o qual implementa uma busca ao registrador de dados da memória informando o endereço ao registrador de endereços da memória (neste caso apontado pelo conteúdo dos registradores CS e IP).

```
While (not terminou) do begin
  {busca a proxima instrucao a ser executada - CICLO FETCH}
  Instrucao:= RDMSegOfs (CS,IP);
  inc(IP);
  case instrucao of
  {chama o procedimento que implementa a instrucao - CICLO DE EXECUTE}
    $00: {.... }
    $F4: {instrução halt - opcode HLT}
          terminou:= true;
         {encerra a execucao qdo encontra halt}
    $FF: {....}
  end;{case}
end; {while}
```

Fonte: Mattos, Tavares e Oliveira (1998, p. 143).

Quadro 8 – Implementação do ciclos de *fetch* e *execute* em versões anteriores do VXt

Uma vez obtida a instrução a ser executada, o registrador IP é incrementado passando a apontar para o próximo byte na memória (o qual pode ser a próxima instrução a ser executada ou, o complemento da instrução atual). Em versões anteriores, a parte final do ciclo de busca (para instruções com mais de 1 byte de comprimento) e o início do ciclo de execução eram implementados como apresentado no Quadro 8. Ou seja, uma enorme estrutura *case*, a partir da qual eram chamados os procedimentos que implementavam respectivamente cada instrução.

Na última versão, substituiu-se a estrutura *case* que possuía 255 entradas, por uma menor, e mais facilmente manipulável. Na realidade, criou-se uma tabela na qual estão armazenadas algumas outras informações comuns a todas as instruções, tais como: tamanho da instrução, e o endereço do procedimento que a implementa. Como algumas instruções necessitavam de parâmetros e outras não, declarou-se um registro de tamanho variável no qual identifica-se através da variável tipo, se o procedimento possui ou não parâmetro. O Quadro 9 apresenta um extrato da tabela.

```
type
  tipo_procedure_sem_parametro = procedure;
  tipo_procedure_com_parametro_byte = procedure (campo1: byte);
  tipo_tab_instrucoes  = record
     tam : byte;  {contem o numero de bytes da instrucao} param: byte; {contem o parametro}
     case tipo : byte of
        0 : (SemParam : tipo_procedure_sem_parametro); {procedimento sem parametro}
        1 : (ComParam : tipo_procedure_com_parametro_byte); {procedimento com parametro}
   end; {record}
const
  tabInstr :array [0..254] of tipo_tab_instrucoes = (
{ opcode                                                    nome do procedimento que implementa a instrucao}
{ ..............  }  ( ...), { ..............}  ( ...), { ..............}  ( ...), { ............}  ( ...), { ...........}  ( ...),
{ $16-PUSH}   (tam:0;  param:$16;  tipo:1; ComParam: Push_SEGREG                                    ),
{ ..............  }  ( ...), { ..............}  ( ...), { ..............}  ( ...), { ............}  ( ...), { ...........}  ( ...),
{ $37- AAA }   (tam:0;  param:$00;  tipo:0; SemParam: Aaa                                           ),
{ ..............  }  ( ...), { ..............}  ( ...), { ..............}  ( ...), { ............}  ( ...), { ...........}  ( ...),
);  {fim da declaração da tabela}
```

Fonte: Mattos, Tavares e Oliveira (1998, p. 143)

Quadro 9 - Declaração da tabela de controle das instruções do processador

Cabe destacar que, esta nova estrutura tornou o código-fonte mais facilmente adaptável, permitiu que novas informações sejam agregadas a futuras implementações do VXt, como por exemplo, o número de ciclos de processador que cada instrução consome. Neste caso, basta incluir um novo campo na declaração do registro, e a correspondente inicialização dos valores na tabela.

Em função desta nova estratégia, o código que implementa os ciclos de busca e execução tornou-se mais simples e, portanto, de mais fácil compreensão e manipulação.

O Quadro 10 demonstra que os procedimentos declarados na tabela apresentada no Quadro 9 são codificados como qualquer procedimento comum na linguagem Pascal e, o Quadro 11 apresenta a nova versão do código, demonstrando como é realizada a chamada de um procedimento declarado dentro de uma tabela. Inicialmente verifica-se se o procedimento possui ou não parâmetros. Em caso negativo, simplesmente faz-se uma referência ao campo `SemParam` da tabela, o que efetivamente ativa o procedimento; por outro lado, se o procedimento possui parâmetros, repete-se o procedimento acima, inserindo-se a informação

de que o procedimento necessita entre parêntesis (como se fosse realizada uma chamada normal de procedimento com parâmetros).

(a)	(b)

Procedure Aaa;
begin
if (lo(A.L) > 9) or Setado(faf)
then begin
A.L:= A.L + 6; inc(A.H);
Seta(faf); Seta(fcf);
A.L:= A.L **and** $F;
end;
end;
	Procedure Push_REG(instrucao: byte);
	var valor: **word**;
	begin
	case instrucao **of**
	$50: valor:= AX;
	.....
	$57: valor:= DI;
	end;
	.....
	end;

Fonte: Mattos, Tavares e Oliveira (1998, p. 144).

Quadro 10 – Exemplo de implementação de procedimento (a) sem parâmetro e (b) com parâmetro

```
While (not terminou) do begin
   {Busca a proxima instrucao a ser executada - CICLO FETCH}
   Instrucao:= RDMSegOfs (CS,IP);
   inc(IP);
   case instrucao of
     {chama o procedimento que implementa a instrucao - CICLO DE EXECUTE}
      . . . {trata algumas excecoes}
      $F4: {instrução halt - opcode HLT}
            terminou:= true;
           {encerra a execucao qdo encontra halt}
   else
     if  tabInstr[instrucao].tipo = semParametro then
        tabInstr[instrucao].SemParam
     else
       tabInstr[instrucao].ComParam(tabInstr[instrucao].param);
   end;{case}
```

Fonte: Mattos, Tavares e Oliveira (1998, p. 144).

Quadro 11 - Ciclos de busca e execução a partir da nova estrutura

Como se pode observar, apesar da implementação da tabela anteriormente descrita, fez-se necessário o uso da estrutura c*ase*, tendo em vista o tratamento diferenciado de alguns procedimentos e mesmo, da identificação da instrução *halt*, a qual encerra a seqüência de buscas e execuções a que o processador é submetido, a partir do momento em que é inicializado.

2.1.3.4 O *chip* PIC 8259A

Interrupções são sinais enviados a CPU pelo hardware, para requisitar a atenção ou alguma ação. O PIC 8259A intercepta os sinais, determina seu nível de importância em relação a outros sinais que ele recebe e interrompe a CPU baseado nesta determinação. A

CPU quando recebe o sinal de interrupção, chama uma rotina, a qual está associada com aquela interrupção em particular.

O PIC 8259A pode controlar até 8 fontes de interrupção numeradas de 0 a 7. As interrupções, que podem ser oriundas de diversos pontos da placa do sistema, são controladas de modo a permitir que somente uma de cada vez interrompa o processador através do pino INTR. Para tanto, cada interrupção possui uma prioridade associada. Quanto menor o número da interrupção, maior é a sua prioridade (interrupção IRQ0 = prioridade máxima).

Se existe uma rotina tratadora (*interrupt handler*) associada a determinada interrupção, então a entrada correspondente na tabela possui o endereço de início da mesma; caso contrário, possui o endereço de uma rotina que somente executa uma instrução IRET, e retorna o controle para o código que estava sendo executado no momento que ocorreu a interrupção.

Cabe destacar que, na versão 2.0, o VXt não implementava completamente a PIC, mas simulava seu funcionamento através da geração aleatória de um sinal que interrompia o processador conforme apresentado anteriormente. Apesar disto, já era possível a execução de instruções INT, ou seja, interrupções geradas por software, o que será explicado a seguir.

2.1.3.5 O mecanismo de tratamento de interrupções

Interrupções e exceções são os dois mecanismos usados para interromper a execução de um programa. Exceções são eventos síncronos que constituem-se em respostas do processador a condições detectadas durante a execução de uma instrução, tais como: tentativa de divisão por zero, ou tentativa de execução de um código de instrução inválido. Interrupções são eventos assíncronos disparados por dispositivos externos para solicitar atendimento do processador.

Uma outra classe de interrupções chamada de interrupção de software, facilita a transferência intencional do controle de fluxo de uma aplicação, valendo-se do mecanismo de interrupções do processador. No processador Intel, estas interrupções são ativadas executando-se a instrução *interruption* (INT).

Interrupções são eventos que ocorrem de forma imprevisível. Independentemente de ser gerada por hardware ou por software, uma interrupção faz com que o controle da CPU seja desviado para uma rotina cujo endereço está armazenado em uma entrada na tabela de vetores

de interrupções. O microprocessador Intel 8086 reserva a parte inferior da memória para armazenar a referida tabela (a qual contém 256 entradas).

Quando ocorre uma interrupção, o conteúdos dos registradores *flags*, CS e IP (nesta ordem) são empilhados, e é atribuído o valor zero aos *flags* IF e TF. A CPU identifica o número da interrupção consultando a PIC e, finalmente , carrega os registradores CS e IP com o conteúdo do vetor de interrupção associado a interrupção ocorrida, fazendo com que o controle seja transferido para a rotina de atendimento.

O processador Intel 8088/86 usa a pilha (*stack*) e uma tabela de vetores de interrupção *Interrupt Vector Table* (IVT) para tratar adequadamente a ocorrência das mesmas. A IVT inicia no endereço físico 0 da memória real e consiste de uma estrutura de dados do tipo matriz. Cada elemento desta matriz consiste de um apontador (endereços na forma de segmento:deslocamento) para uma rotina especializada no atendimento da interrupção identificada pelo índice da entrada na matriz.

Quando o tratamento da interrupção termina, uma instrução IRET é usada para retornar o controle ao ponto original, ou seja, para carregar novamente o conteúdo dos registradores CS, IP e *flags* com seus valores anteriores a ocorrência da interrupção.

O Quadro 12 apresenta o trecho de código do simulador que trata um sinal de interrupção. Como pode-se observar, na realidade para o hardware virtual, uma interrupção não é um evento assíncrono, uma vez que, ao término do ciclo de execução de cada instrução, é realizada uma consulta ao modulo que implementa o processador de interrupções, para consultar seu estado (se sinalizando a ocorrência de uma interrupção ou não).

```
While (not terminou) do begin
   If (statusPIC) then
      If (InterruptFlagOn) then begin
         SalvaCSIPFlags; SetaFlagsIFTF;
         IP := PegaIP_TVI(pegaPICNumInt); {busca novos valores p/CS e IP na tab.vetores de interrup.}
        CS := PegaCS_TVI(pegaPICNumInt);
       End;
   {continua o fluxo de execução como se não tivesse ocorrido interrupção}
   Instrucao:= RDMSegOfs (CS,IP); {busca a proxima instrucao a ser executada - CICLO FETCH}
   .....
end; {while}
```

Fonte: Mattos, Tavares e Oliveira (1998, p. 146).

Quadro 12 – Ciclos de busca e execução no VXt

Apesar disto, analisando-se a questão do ponto de vista do código que a CPU virtual está executando, tanto o conceito de indivisibilidade das instruções quanto o conceito de assincronismo dos eventos interrupções são totalmente aplicados na medida em que, mesmo no simulador não se pode afirmar em que momento a PIC irá sinalizar a ocorrência de uma

interrupção. Isto porque o módulo que a implementa, usa o gerador de números randômicos para aleatoriamente sinalizar a ocorrência ou não de uma interrupção.

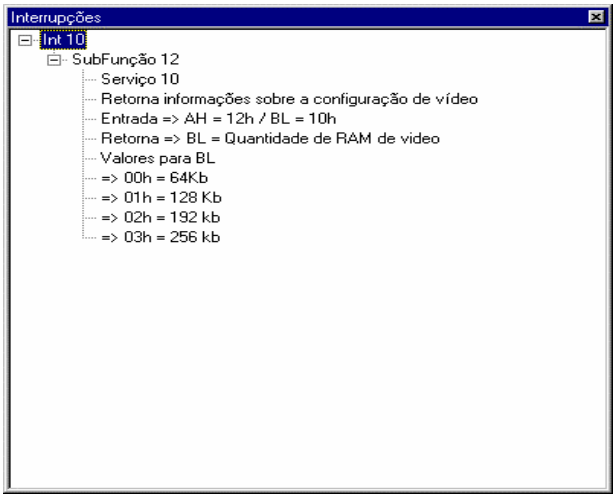
2.1.3.6 Filtro de Interrupções

Uma outra característica importante que o VXt apresenta é a possibilidade de identificação e mapeamento das interrupções por software geradas pela aplicação sendo monitorada. O Quadro 13 apresenta um trecho do código-fonte do procedimento que dispara o tratamento para cada uma das interrupções e a Figura 8 apresenta o resultado deste mapeamento.

```
Procedure
Interrupcao(Interrup :
Byte);
Begin
  Case Interrup Of
      . . . . . .
    21 : Int_21;
     . . . . . .
  End;
End;
```

Fonte: Mattos, Tavares e Oliveira (1998, p. 147).

Quadro 13 - Filtro de interrupções do VXt



Fonte: Mattos, Tavares e Oliveira (1998, p. 147).

Figura 8 – Exemplo de filtro de interrupção

Dessa forma, é possível (desde que configurado para funcionar desta maneira) ao aluno identificar e analisar que informações estão sendo passadas como parâmetro para o núcleo do sistema operacional, e o que o sistema operacional está retornando para a aplicação.

Além de identificar o conteúdo dos parâmetros, o VXt apresenta a descrição do funcionamento da interrupção, bem como o significado dos parâmetros de entrada e o significado das informações retornadas.

2.1.3.7 O detalhamento das instruções

Uma instrução de hardware do processador Intel 8086 pode ocupar entre 1 a 6 bytes de comprimento, mas o código da operação, ou seja, o padrão binário que identifica uma instrução, ocupa sempre os 6 primeiros bits do primeiro byte da mesma. Os outros 2 bits são usados para identificar respectivamente, se a instrução manipula operandos com ou sem sinal e, se a instrução manipula operandos do tipo byte ou *word* (2 bytes).

Para aquelas instruções que ocupam 2 ou mais bytes, o segundo byte é usado para identificar o modo de endereçamento sendo utilizado. Para tanto, este byte é dividido em 3 campos, os quais são descritos na Quadro 14.

Modo
Ocupa 2 bits e indica se a operação envolve operandos em registradores ou em memória (neste caso, mais 2 bytes são usados para indicar o endereço de deslocamento do operando na memória)

Reg
Ocupa 3 bits e identifica o destino da operação. A tabela abaixo apresenta a codificação usada para representar os registradores da CPU.

	W=1	W=0
000	AX	AL
001	CX	CL
010	DX	DL
011	BX	BL
100	SP	AH
101	BP	CH
110	SI	DH
111	DI	BH

RM
Ocupa os restantes 3 bits, e identifica o registrador de origem dos dados ou, o endereço do operando na memória. A tabela abaixo apresenta a codificação usada.

000	DS:[BX+SI+disp]
001	DS:[BX+DI+desl]
010	DS:[BP+SI+desl]
011	DS:[BP+dI+desl]
100	DS:[SI+desl]
101	DS:[DI+desl]
110	DS:[BP+desl]
111	DS:[BX+desl]

Fonte: Mattos, Tavares e Oliveira (1998, p. 148).
Quadro 14 - Descrição dos campos do 2º byte das instruções

Para finalizar esta seção, cabe observar que, as instruções de acesso as portas de I/O (espaço de endereçamento que permite ao processador comunicar-se com os periféricos e vice-versa) não foram implementados nesta versão do VXt.

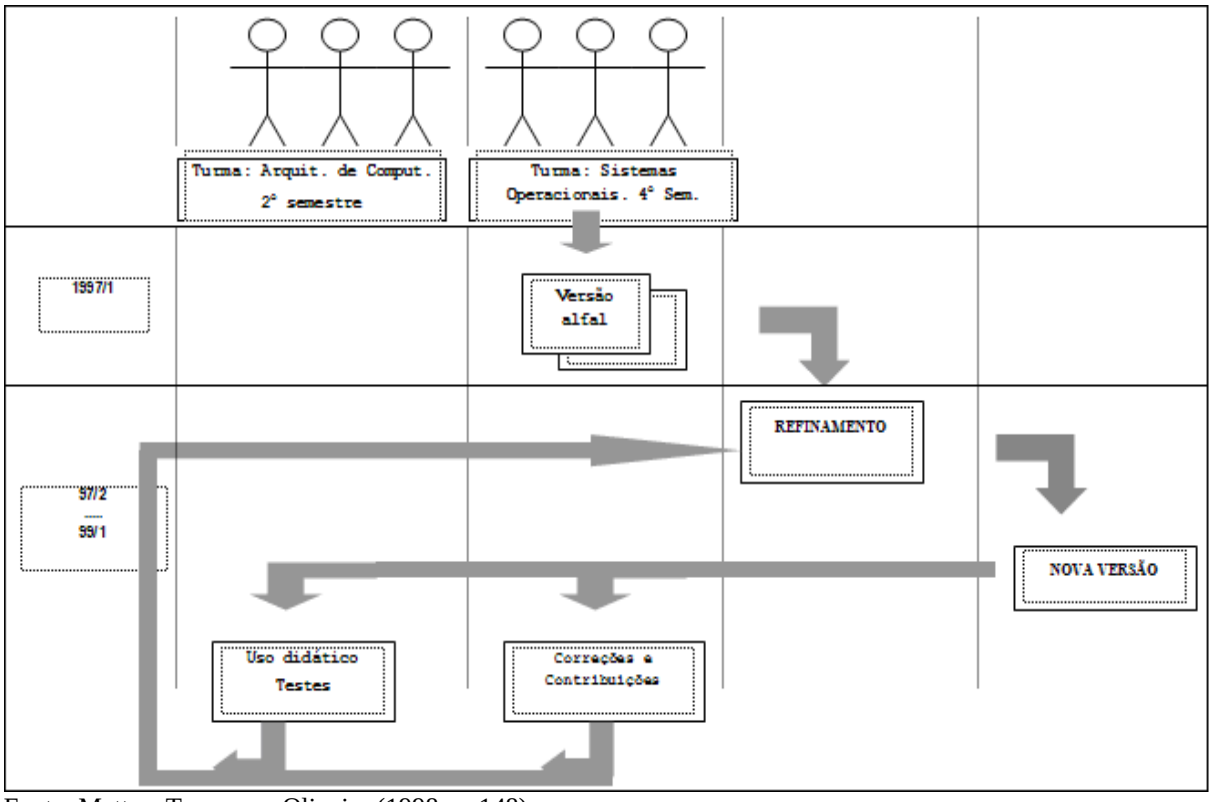
Naturalmente, associado a codificação binária, existe uma representação simbólica. Para cada instrução portanto, há um símbolo denominado mnemônico, o qual descreve "em mais alto nível", o significado da instrução. A Figura 9 apresenta a janela que descreve o mnemônico da instrução atualmente sendo executada pelo VXt.

Fonte: Mattos, Tavares e Oliveira (1998, p. 148).
Figura 9 - Identificação do mnemônico da instrução

2.1.4 A metodologia de desenvolvimento

A publicação seguinte (MATTOS; OLIVEIRA, 1999) acrescentou uma descrição sobre a metodologia de desenvolvimento do projeto caracterizando a integração das atividades em sala de aula.

Adotou-se uma metodologia de aprendizado ativa (Figura 10), através de sucessivos refinamentos, na qual os conceitos básicos seriam implementados gradualmente, e a medida em que o projeto avançasse, maiores níveis de abstração seriam obtidos, sem a perda do que já foi assimilado. Um requisito importante foi o de que a cada semestre, os novos alunos incorporados ao projeto teriam acesso a toda documentação (especificação + código-fonte da aplicação) disponível de tal forma a revisar o que já havia sido construído e continuar o desenvolvimento do projeto.



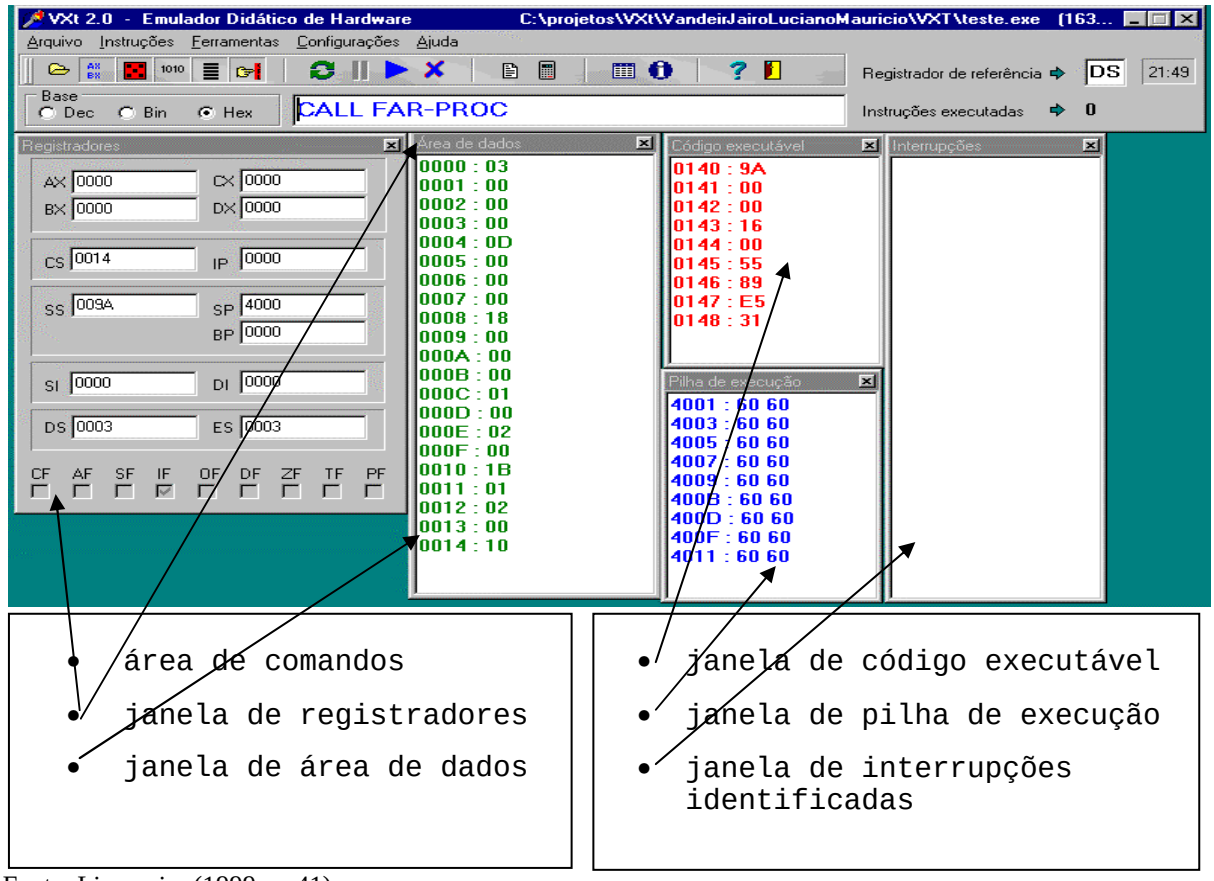
Fonte: Mattos, Tavares e Oliveira (1998, p. 148).

Figura 10 - Ciclo de desenvolvimento do projeto

Paralelamente ao desenvolvimento do plano de ensino das respectivas disciplinas, os alunos foram sendo engajados ao projeto de implementação, de tal forma a desencadear um processo de reflexão e tomada de consciência de seu próprio processo de aprendizagem.

A partir da liberação da primeira versão em meados de 1997, deu-se o início ao ciclo de testes (correções) e incorporações de novos módulos tanto de software como de hardware (virtual).

A ferramenta passou a ser usada pelas turmas de 2º semestre, atuando como auxiliar ao aprendizado dos conceitos de aritmética binária, componentes de hardware e linguagem a*ssembly*. Erros encontrados durante o uso eram reportados para as turmas responsáveis pela implementação, para correção. Durante os 3 primeiros semestres, o processo não apresentou um rendimento muito rápido tendo em vista que as turmas ainda não estavam ambientadas com os conceitos e a filosofia de desenvolvimento do projeto. A partir do momento em que as turmas de 2º semestre (que já haviam tido contato com a ferramenta) começaram a chegar ao 4º semestre, o processo de desenvolvimento passou a ser mais rápido. A Figura 11 apresenta a interface com o usuário da versão 2.0 do VXt.



Fonte: Linzmeier (1999, p. 41).

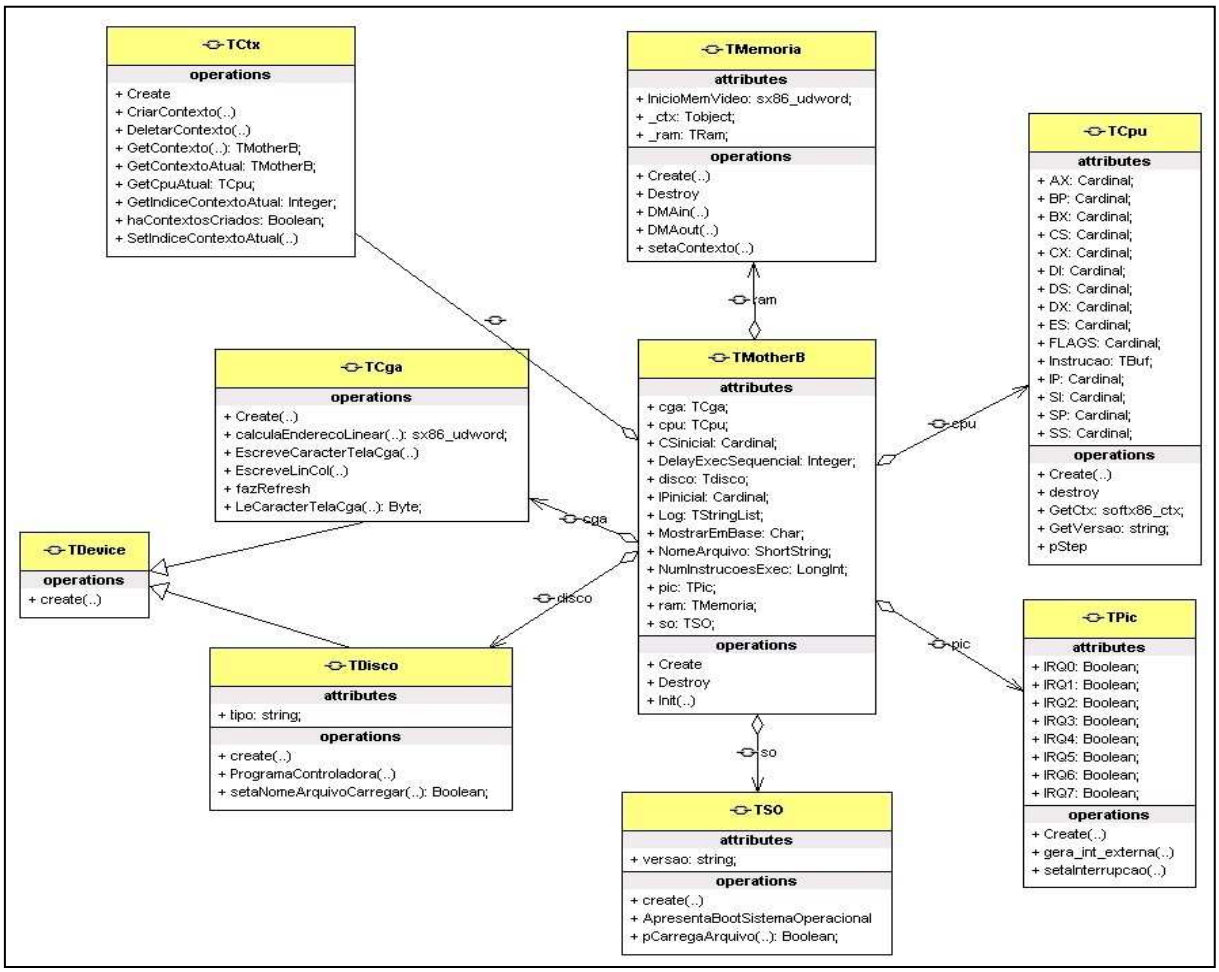
Figura 11 – Características da interface com o usuário

2.1.5 A versão VXt 2004

No trabalho publicado em 2004 (MATTOS et al., 2004), foram acrescentadas informações sobre um processo de reengenharia.

> Não obstante o caráter de aprendizagem de como os componentes internos de um processador funcionam estarem sendo implementados, havia um aspecto frustrante: o VXt não podia executar completamente programas compilados com compiladores tradicionais tendo em vista que, não havia um sistema operacional instalado na máquina virtual. Assim, toda chamada de sistema (que no MS-DOS ocorre através da instrução INT 21h) fazia com que o programa do aluno encerrasse sua execução de forma anormal. (MATTOS, et al., 2004).

No segundo semestre de 2003, iniciou-se um processo de reengenharia no projeto, de tal forma a substituir o código que implementava as instruções por um equivalente desenvolvido no projeto X86 (SOURCEFORGE, 2003 apud MATTOS et. al., 2004). Dessa forma, foi possível conjugar os esforços no sentido de obter-se um simulador fiel, e de desenvolver-se uma ferramenta didática. Além disso, o projeto foi totalmente remodelado, sendo estruturado conforme o diagrama de classes apresentado na Figura 12.



Fonte: Mattos et al. (2004).

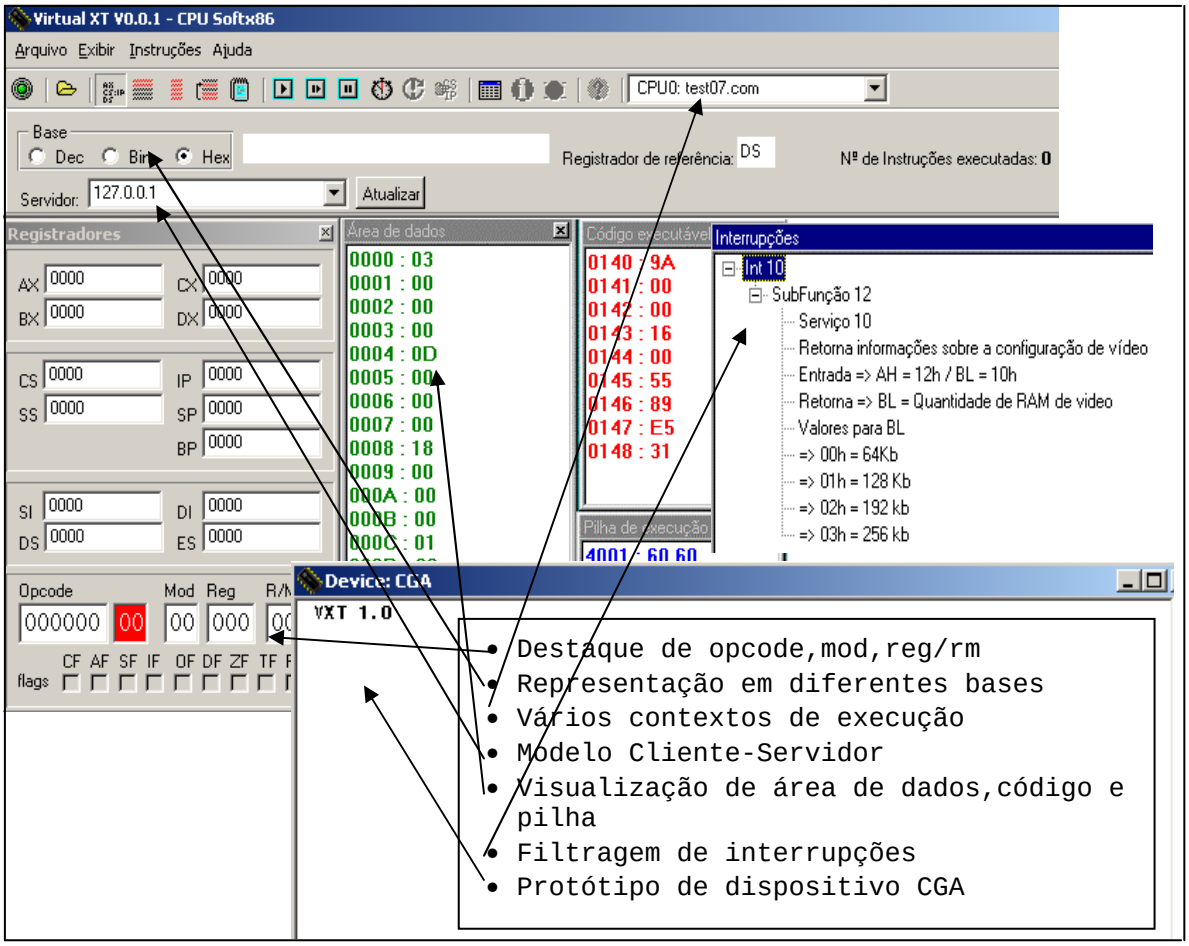
Figura 12 – Diagrama de Classes VXt

Assim sendo, o VXt passou a apresentar uma nova arquitetura. Esta versão apresenta algumas características importantes em relação às versões anteriores, quais sejam (MATTOS, et al., 2004):

a) modelo cliente-servidor;
b) isolamento entre a interface com o usuário e a implementação do processador;
c) possibilidade de execução de vários exemplos em paralelo através do congelamento de contextos;
d) possibilidade de armazenar e resgatar o contexto de execução de um programa em outro momento;
e) possibilidade de salvar o "*log*" de execução do programa em arquivo para posterior análise.

A arquitetura baseada no modelo cliente-servidor permitiu que fosse possível ao professor disparar o servidor (que implementa o processador), e sua instância da interface com o usuário (Figura 13).

Também concede aos alunos dispararem suas instâncias de interface com o usuário, para "assistirem" a execução passo a passo conduzida pelo professor. Esta arquitetura permite que, em determinado momento, trabalhos práticos sejam conduzidos dentro de pequenos grupos, onde o líder do grupo dispara o módulo servidor, e seus colegas conectam-se a ele através do endereço IP do servidor.



Fonte: Mattos et al. (2004)

Figura 13 – Modelo da interface cliente-servidor

Este processo possibilita as seguintes variações (MATTOS, et al., 2004):

a) o professor conduz a aula e os alunos acompanham a execução;

b) o professor dispara um trabalho em grupo e pode conectar-se na estação do líder para acompanhar o andamento do trabalho;

c) cada aluno individualmente pode executar sua própria cópia do servidor e da interface cliente em uma máquina local podendo executar exercícios individuais.

Como resultado do processo de reengenharia, foi possível dispararem-se novas frentes de desenvolvimento em paralelo graças a instalação de um servidor CVS. O CVS é um sistema de controle de versões concorrente que possibilita o trabalho paralelo de várias equipes.

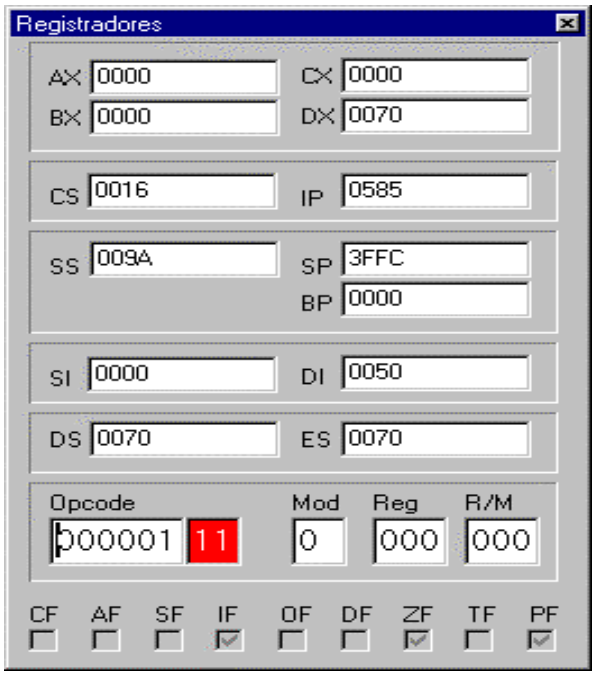
Esta versão do VXt apresenta as seguintes características (MATTOS et al., 2004):

a) a atual versão do VXt apresenta uma interface de baixo nível tendo em vista que, um dos objetivos da proposta VXt é o de permitir ao aluno manipular código executável gerado pelos compiladores da máquina real;

b) a implementação do hardware busca, na medida do possível, aproximar-se com

máxima fidelidade à implementação do hardware real. Em função disto, sua aplicabilidade torna-se eminentemente didática, na medida em que é possível utilizá-lo para o ensino de conceitos básicos de linguagem de programação *assembly*, conceitos de hardware e alguns conceitos básicos de sistemas operacionais;

c) possui uma interface com o usuário (Figura 13) bastante amigável;
d) apresenta informações sobre o passo de relocação de arquivos .EXE tanto ao nível de *header* quanto ao nível de tabela de relocação;
e) permite a manipulação de arquivos binários puros, o que possibilita a construção de estruturas muito simples para teste de algum conceito/instrução sem a necessidade de elaboração de um programa completo;
f) permite a alteração da base numérica para representação das informações na tela, sendo possível optar-se por: decimal, hexadecimal e binário;
g) apresenta um *log* da seqüência de execução realizada, de tal forma que o aluno pode analisar o fluxo de execução de um determinado programa ou rotina, mesmo após o término da execução;
h) a medida em que instruções do tipo INT são detectadas, as mesmas são filtradas e apresentadas em uma janela específica (Figura 13), sendo possível ao aluno analisar efetivamente o que ocorre. São apresentados: a descrição da função, os parâmetros de entrada e o valor de retorno da mesma;
i) permite a análise por parte do aluno, do *opcode* da instrução sendo executada (Figura 14);
j) apresenta dicas explicativas (*hints*) sobre os conceitos teóricos que sustentam todo o funcionamento da CPU bem como das instruções de máquina;
k) identifica quais são os registradores de referência; clicando-se na área de código destacam-se os registradores CS:IP, clicando-se na janela de área de pilha, destacam-se os registradores SS:SP, e assim por diante, induzindo desta forma o aluno ao entendimento da função dos registradores de segmento e deslocamento.



Fonte: Mattos, Tavares e Oliveira (1998, p. 140).

Figura 14 - Janela de registradores e *opcode* do VXt

2.1.6 A versão base para a implementação do VXt-C/S

A versão utilizada para a implementação do VXt-C/S apresenta as seguintes características em relação aquelas relacionadas na publicação em 2004 (MATTOS et al., 2004):

a) separação das funções cliente e servidor que estavam implementadas no mesmo fonte fazendo com que a versão em Delphi seja responsável por operar o software e propagar as alterações de tela para o *middleware*;
b) inclusão de um conjunto de meta-forms para viabilizar a separação entre a versão em Delphi e o *middleware*.

2.2 SOFTX86

Um emulador é um software que viabiliza a execução de aplicações escritas para um processador em outro, traduzindo o conjunto de instruções do processador sendo emulado por instruções do processador alvo. O VXt enquadra-se nesta categoria uma vez que implementa

o conjunto completo de instruções de um processador Intel 8086 (embora execute em processadores da arquitetura Intel).

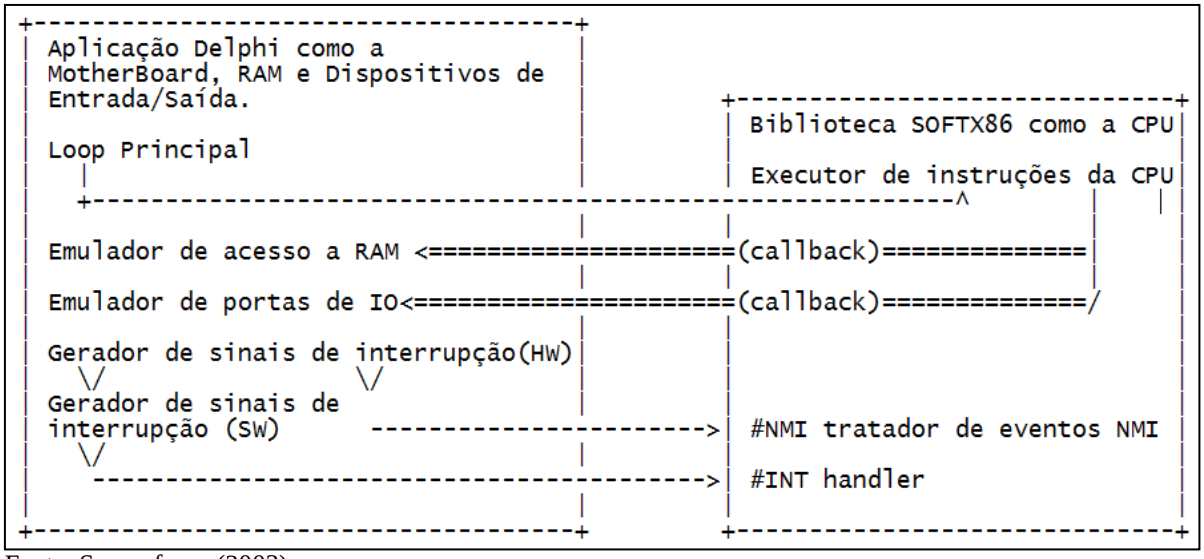
A próxima seção apresenta uma descrição da estrutura da biblioteca softx86 e descreve a integração entre o VXt e esta biblioteca.

2.2.1 Descrição da arquitetura da biblioteca softx86

Segundo (SOURCEFORGE, 2003) a biblioteca Softx86 foi concebida de forma a viabilizar a construção de programas de emulação de hardware. A biblioteca emula somente a CPU. A simulação dos demais componentes de hardware deve ser realizada por um programa externo. No presente projeto, o VXt é responsável pela construção de toda a infra-estrutura adicional (*motherboard*, memória, espaço de endereçamento de I/O, periféricos, etc.) (Figura 15).

Entre as principais características estão (SOURCEFORGE, 2003):

a) precisão na emulação da CPU;
b) tamanho da memória ajustável;
c) projeto que modela o comportamento dos seguintes sinais de hardware (na forma de *callbacks* no programa hospedeiro) :
   - clock,
   - acesso ao espaço de I/O,
   - acesso ao espaço de endereçamento da memória,
   - interrupções por software,
   - interrupções de hardware.



Fonte: Sourceforge (2003).

Figura 15 – Integração entre o VXt e a biblioteca Softx86

2.2.1.1 Contexto da CPU

Uma outra característica importante é que a API da biblioteca Softx86 utiliza estruturas de contexto para representar a CPU. Estas estruturas de contexto são declaradas na aplicação hospedeira (VXt). Com isto é possível emular mais de uma CPU simultaneamente utilizando a mesma biblioteca.

Esta estratégia possibilita que todo o estado da CPU seja compartilhado tanto pela biblioteca como pela aplicação hospedeira.

No VXt, esta estrutura foi declarada utilizando as definições de tipos apresentadas no Quadro 15, tendo em vista compatibilizar a denominação utilizada em C e em Delphi.

```
Type
//Shortint  -128..127   signed 8bit
      sx86_sbyte = shortint;
//Byte      0..255      unsigned 8-bit
      sx86_ubyte = Byte;
//Smallint  -32768..32767     signed 16-bit
      sx86_sword = SmallInt;
//Word      0..65535    unsigned 16-bit
      sx86_uword = Word;
//Longint   -2147483648..2147483647 signed 32-bit
      sx86_sdword = longint;
//Longword  0..4294967295     unsigned 32-bit
      sx86_udword = longword;
//Int64     -2^63..2^63-1     signed 64-bit
      sx86_uldword = int64;
      sx86_sldword = int64;
```

Quadro 15 – Compatibilidade de tipos entre Delphi e API Softx86

A partir destas definições de tipo, foram declaradas as estruturas auxiliares apresentadas no Quadro 16, Quadro 17 e Quadro 18.

```
Type
 //estrutura utilizada para acesso a 16bits dos registradores de 32 bits
  __dummy_w = record
    lo : SX86_UWORD; {lower 16 bits of register (i.e. AX) }
    hi : SX86_UWORD; {upper 16 bits of register }
  end;
//estrutura utilizada para acesso a 8 bits dos registradores de 16 bits
  __dummy_b = record
    lo : SX86_UBYTE; {lower 8 bits of register (i.e. AL) }
    hi : SX86_UBYTE; {bits 8-15 of register (i.e. AH) }
    extra: array[0..1] of SX86_UBYTE;
  end;


 softx86_regval = record
  case byte of
   1:(val:SX86_UDWORD;);{the entire 32-bit register as a whole(i.e. EAX)
}
   2:(w : __dummy_w;);
   3:(b : __dummy_b;);
  end;
```

Quadro 16 – Declaração de estruturas auxiliares para acesso a partes de registradores da CPU

Para facilitar o acesso ao conjunto de registradores, a biblioteca utiliza uma enumeração conforme apresentado no Quadro 17.

```
Type
      softx86_genregidx16 = (
          SX86_REG_AX{=0},
          SX86_REG_CX,
          SX86_REG_DX,
          SX86_REG_BX,
          SX86_REG_SP,
          SX86_REG_BP,
          SX86_REG_SI,
          SX86_REG_DI);
        softx86_segregidx = (
          SX86_SREG_ES{=0},
          SX86_SREG_CS,
          SX86_SREG_SS,
          SX86_SREG_DS,
          SX86_SREG_Dummy1,
          SX86_SREG_Dummy2,
          SX86_SREG_Dummy3,
          SX86_SREG_Dummy4 );
```

Quadro 17 – Declaração de identificadores de acesso aos registradores da CPU

A partir das definições acima, o contexto da CPU é declarado em Delphi como apresentado no Quadro 18.

```
Type
 softx86_cpustate = record
  {VISIBLE REGISTERS }
  {general purpose registers}
  general_reg: Array[SX86_REG_AX..SX86_REG_DI] of SOFTX86_REGVAL;
  {segment registers}
  segment_reg: Array[SX86_SREG_ES..SX86_SREG_Dummy4] of SOFTX86_SEGREGVAL;
  {instruction and FLAGS registers}
  reg_flags: SOFTX86_REGVAL;
  reg_ip: SX86_UDWORD;
  {control/machine state registers}
  reg_cr0: SX86_UDWORD;
  {INVISIBLE REGISTERS (FOR CACHING PURPOSES)}
  {instruction pointer saved before execution. These are copied from CS:IP
       before executing an instruction}
  reg_cs_exec_initial: SX86_UWORD;
  reg_ip_exec_initial: SX86_UDWORD;
  {instruction pointer for decompiler}
  reg_cs_decompiler: SX86_UWORD;
  reg_ip_decompiler: SX86_UDWORD;
  {interrupt flags}
  int_hw_flag: SX86_UBYTE;{flag to indicate hardware (external) interrupt
pending}
  int_hw: SX86_UBYTE;{if int_hw_flag==1, which interrupt is being signaled}
  int_nmi: SX86_UBYTE;{flag to indicate NMI hardware (external) interrupt
pending}
{other internal flags}
  rep_flag: SX86_UBYTE;          {0=no REP loop, 1=REPNZ, 2=REPZ }
  is_segment_override: SX86_UBYTE;{1=segment override was encountered }
  segment_override: SX86_UWORD;{value of segment that overrides the default}
end {softx86_cpustate};
```

Quadro 18 – Definição do estado da CPU a partir da especificação da biblioteca Softx86

2.2.1.2 Criação de uma CPU e conexão com a aplicação hospedeira.

Para criar uma instância de CPU, inicialmente aloca-se espaço em memória para o contexto da CPU e inicializam-se os procedimentos de *callbac*k que a biblioteca fará uso. O Quadro 19 apresenta a estrutura de *callbacks* implementada no VXt.

```
procedure pOn_read_memory(_ctx : pointer;
                          address : sx86_udword;
                          var buf : sx86_ubyte;
                          size : Integer); cdecl;

procedure pOn_Write_Memory(_ctx : pointer;
                          address : sx86_udword;
                          var buf : sx86_ubyte;
                          size : Integer); cdecl;

//called when an external (to this library) interrupt occurs
procedure pOn_hw_int(_ctx:pointer;i: sx86_ubyte);cdecl;

//called when an internal (emulated software) interrupt occurs }
procedure pOn_sw_int(_ctx:pointer;i: sx86_ubyte);cdecl;

//called when an internal (emulated software) interrupt occurs }
procedure pOn_sw_int_ack(_ctx:pointer;i: sx86_ubyte);cdecl;

//called when the CPU (this library) acknowledges the interrupt }
procedure pOn_hw_int_ack(_ctx:pointer;i: sx86_ubyte);cdecl;

//called when an external (to this library) NMI interrupt occurs }
procedure pOn_nmi_int(_ctx:pointer);cdecl;

//called when the CPU (this library) acknowledges the NMI interrupt }
procedure pOn_nmi_int_ack(_ctx:pointer);cdecl;

//called when reading an I/O port */
procedure pOn_read_io(_ctx:pointer;address: sx86_udword;
                      var buf:sx86_ubyte;size:integer);cdecl;
//called when writing an I/O port */
procedure pOn_write_io(_ctx:pointer;address: sx86_udword;
                        var buf:sx86_ubyte;size: integer);cdecl;
```

Quadro 19 – Definição dos procedimentos de *callback* no VXt

A criação de uma CPU no contexto do VXt é apresentada no Quadro 20.

```
constructor TCpu.Create(mb:TObject);
var
   strDecodPrimeiraInstrucao:string;
begin
  inherited Create;

  ocorreuInstrucaoInvalida := false;
  cpuEmHalt     := false;

  motherboard := mb;
  wCpu := self;
  //inicializa o contexto da CPU
  if softx86_init(wCtx, SX86_CPULEVEL_8086 ) <> 1 then
//0 (zero)Initialization failed, insufficient memory, or CPU revision not supported.
//1 Initialization succeeded
//2 Initialization succeeded, CPU revision requested grossly incomplete and buggy.
  begin
    exit;
  end;
  { Inicializa contexto }
  softx86_set_instruction_ptr(wCtx, $1000, $100);//inicializa IP
  softx86_set_stack_ptr(wCtx,$1000,$FFF8); //inicializa pilha
  softx86_setsegval(wCtx, ord(SX86_SREG_DS),$1000);
  softx86_setsegval(wCtx, ord(SX86_SREG_ES),$1000);
  wCtx.state.reg_flags.val := 512;   //inicializa com as interrupçòes habilitadas

  { Aponta as callbacks que fazem o acesso a memória }
  wCtx.callbacks.on_read_memory := pOn_Read_Memory;
  wCtx.callbacks.on_write_memory := pOn_write_memory;

  { Aponta as callbacks que fazem o controle de interrupcoes }
  wCtx.callbacks.on_hw_int := pOn_hw_int;
  wCtx.callbacks.on_sw_int := pOn_sw_int;

  wCtx.callbacks.on_sw_int_ack := pOn_sw_int_ack;
  wCtx.callbacks.On_hw_int_ack :=  pOn_hw_int_ack;
  wCtx.callbacks.On_nmi_int :=  pOn_nmi_int;
  wCtx.callbacks.On_nmi_int_ack :=  pOn_nmi_int_ack;
  wCtx.callbacks.on_read_io := pOn_read_io;
  wCtx.callbacks.on_write_io := pOn_write_io;
end;
```

Quadro 20 – Criação de uma instância de CPU e inicialização do contexto e dos procedimentos de *callback*

2.2.1.3 O funcionamento da CPU

O Quadro 21 apresenta o código do VXt que, interagindo com a biblioteca softx86, viabiliza a execução de uma instrução de máquina recuperada a partir do endereço de memória apontado pelo par de registradores CS:IP. A função `softx86_decompile_exec_cs_ip(wctx)` é utilizada para obter uma representação textual da instrução a ser executada para fins de visualização. A função `softx86_step(wCtx)` é quem efetivamente executa uma instrução atualizando o contexto de execução da CPU.

```
procedure TCpu.pStep;
begin
  if ocorreuInstrucaoInvalida or cpuEmHalt then
     exit; //nao deixa executar mais!

  // decompila a instrucao apontada por CS:IP para apresentar na tela
  softx86_decompile_exec_cs_ip(wctx);

  if softx86_step(wCtx) = 0 then
//0 (zero) Execution failed. This will most likely happen if the simulated CPU
// encountered an instruction that it didn't recognize, if any member of the structure
// is invalid,or if ctx == NULL. CS:IP are reset to point to the offending instruction
// in this case.
// 1       Execution succeeded, CS:IP changed, everything OK.
  begin
     showmessage('Erro: a CPU falhou em executar ou reconhecer a instrucao');
     ocorreuInstrucaoInvalida := true; //sinaliza a ocorrencia de instrução inválida.
     Exit;
  end;
  else //atualiza os campos do formulário de registradores
    begin
      wAX    := wctx.state.general_reg[SX86_REG_AX].val;
      wBX    := wctx.state.general_reg[SX86_REG_BX].val;
      wCX    := wctx.state.general_reg[SX86_REG_CX].val;
      wDX    := wctx.state.general_reg[SX86_REG_DX].val;
      wDI    := wctx.state.general_reg[SX86_REG_DI].val;
      wSI    := wctx.state.general_reg[SX86_REG_SI].val;
      wBP    := wctx.state.general_reg[SX86_REG_BP].val;
      wSP    := wctx.state.general_reg[SX86_REG_SP].val;
      wFlags := wCtx.state.reg_flags.val;
      wDS    := wCtx.state.segment_reg[SX86_SREG_DS].val;
      wSS    := wCtx.state.segment_reg[SX86_SREG_SS].val;
      wCS    := wCtx.state.segment_reg[SX86_SREG_CS].val;
      wIP    := wCtx.state.reg_ip;
      wES    := wCtx.state.segment_reg[SX86_SREG_ES].val;
    end;
    if (wCtx.state.reg_flags.val and SX86_CPUFLAG_INTENABLE) = 0  then //$00000200;
    begin //interrupcoes habilitadas ... pode atender
      softx86_ext_hw_ack(wCtx);  //manda um ack para a pic!!!!
    end;
end;
```

Quadro 21 – Funcionamento de um passo de execução da CPU

2.2.1.4 O acesso a memória do VXt

O acesso à memória é realizado através de funções *callback*. O Quadro 22 apresenta o código do procedimento para acesso de leitura e de escrita à memória do VXt (implementada fora da biblioteca Softx86).

```
//-------------------------------------------------------------------------------------
//  CALLBACK:  PON_READ_MEMORY
//-------------------------------------------------------------------------------------
//Funcao: Callback que lê um conteúdo na memória
//-------------------------------------------------------------------------------------
procedure pOn_read_memory(_ctx : pointer; address : sx86_udword; var buf : sx86_ubyte;
size : Integer); cdecl;
begin
  if address < ctx.GetContextoAtual.ram.InicioMemVideo then
    copymemory(@buf,@ctx.GetContextoAtual.ram._ram[address],size)
  else
    buf := ctx.getContextoAtual.cga.leCaracterTelaCGA(address);
end;

//-------------------------------------------------------------------------------------
//  CALLBACK:  PON_WRITE_MEMORY
//-------------------------------------------------------------------------------------
//Funcao: Callback que escreve na memória
//-------------------------------------------------------------------------------------
procedure pOn_Write_Memory(_ctx : pointer; address : sx86_udword; var buf : sx86_ubyte;
size : Integer); cdecl;
begin
  if address < ctx.GetContextoAtual.ram.InicioMemVideo then
    copymemory(@ctx.GetContextoAtual.ram._ram[address],@buf,size)
  else
    ctx.getContextoAtual.cga.escreveCaracterTelaCGA(address,buf);
end;
```

Quadro 22 – Declaração dos procedimentos *callback* para acesso à memória

Um aspecto que merece destaque no código apresentado no Quadro 22 é a necessidade explícita de verificação sobre a tentativa de acesso a memória de vídeo. Caso o acesso a memória seja destinado a faixa de endereços onde reside a memória de vídeo, são chamados os procedimentos `leCaracterTelaCGA` e `escreveCaracterTelaCGA` respectivamente para ler o conteúdo de uma coordenada de tela e escrever o conteúdo em uma coordenada de tela.

2.3 PADRÃO MVC

Os padrões de projeto de software, descrevem soluções para problemas recorrentes no desenvolvimento de sistemas de software orientados a objetos. Um padrão de projeto estabelece um nome e define o problema, a solução, quando aplicar esta solução e suas consequências (PADRÕES, 2006).

Segundo Braz (2006) “quando utilizamos padrões, estamos levando em conta experiências de outros projetos de desenvolvimento, aumentando assim as chances de chegarmos em uma solução correta pois erros passados poderão ser evitados.”

Padrões de projeto trazem inúmeras vantagens na modelagem e implementação de um software (BRAZ, 2006):

a) possibilidade de projetar soluções mais rapidamente e com qualidade já que os

padrões são soluções comprovadamente eficientes para problemas já conhecidos;

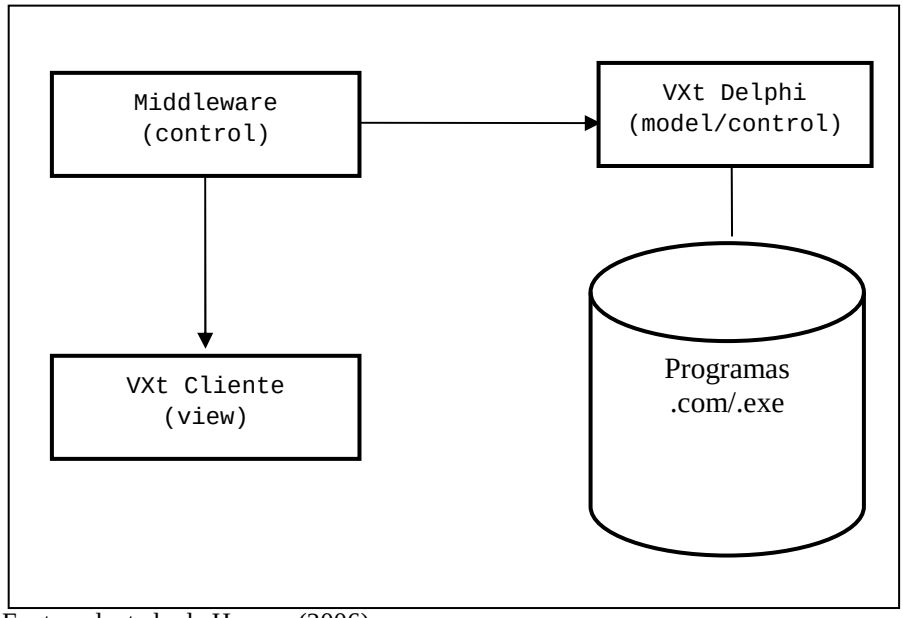
b) visam principalmente flexibilidade, organização e reaproveitamento de código, o que resulta em maior produtividade, qualidade e facilidade de manutenção das aplicações assim desenvolvidas.

A medida que as aplicações e os sistemas crescem, é importante dividir e redividir a responsabilidade, a fim de que classes e pacotes permaneçam suficientemente pequenos para a sua manutenção (METSKER, 2004, p. 98).

Uma solução dentre os padrões de projeto é o padrão MVC que tem por objetivo dividir os elementos de um sistema em três tipos de camadas: modelo (*Model*), visão (*View*) e controlador (*Controller*) (GAMMA et al., 1994).

Fundão da Computação (2004) diz “MVC é útil principalmente para aplicações grandes e distribuídas onde dados idênticos são visualizados e manipulados de formas variadas”.

Segundo Fundão da Computação (2004) as visões podem usar o modelo de dados para exibir resultados, mas elas não são responsáveis por atualizar as informações. Os controladores são responsáveis por fazer alterações no modelo de dados. O modelo, por sua vez, é responsável por representar os dados da aplicação. Algumas vezes o modelo de dados também inclui a lógica de negócio e algumas vezes a lógica de negócio existe na camada do controlador. A Figura 16 ilustra o padrão MVC para o VXt.



Fonte: adaptado de Hansen (2006).

Figura 16 – Padrão MVC

O padrão MVC oferece algumas características que fazem dele uma boa escolha de padrão a ser utilizado no desenvolvimento de software, dentre as quais, tem-se (SILVEIRA,

2006, p. 29):

a) separar dados (*Model*) da interface do usuário (*View*) e do fluxo da aplicação (*Controller*);
b) permitir que uma mesma lógica de negócio possa ser acessada e visualizada através de várias interfaces;
c) a lógica de negócio é independente da camada de interface com o usuário (*View*).

No contexto do projeto VXt, o padrão MVC foi adotado na seguinte perspectiva:

a) a versão em Delphi implementa o modelo do emulador do PC-XT e o modelo do controlador (uma vez que, o controle da aplicação ainda permanece na aplicação Delphi);
b) o *middleware* implementa parte do controlador da aplicação (uma vez que é o módulo responsável pela propagação das informações para os clientes e também por propagar os comandos de abertura/fechamento de janelas);
c) o módulo de interface do aluno implementa a camada de visão.

2.4 ARQUITETURA CLIENTE/SERVIDOR

Uma arquitetura cliente/servidor é uma abordagem da computação que separa os processos em plataformas independentes interagindo entre si, permitindo que os recursos sejam compartilhados enquanto se obtém o máximo de benefício de cada dispositivo diferente (BOCHENSKI, 1995, p. 8).

Os componentes básicos dos sistemas cliente/servidor são: parte cliente, parte servidor e uma parte de rede. O cliente faz a requisição e o servidor atende à requisição do cliente (BOCHENSKI, 1995, p. 8).

Para Doerner (2004, p. 17), “a arquitetura cliente/servidor estabeleceu um novo paradigma de processamento de dados, diversificando o processamento entre dois processos de software distintos (cliente e servidor)”. Ao mesmo tempo a arquitetura visa fornecer recursos que coordenem estes processos de forma que, a perda de sincronização, não resulte em alterações ou perda de informações para o sistema.

Segundo Bochenski (1995, p. 9), há 5 características obrigatórias para que um sistema possa ser definido como cliente/servidor:

a) uma arquitetura cliente/servidor consiste em um processo de cliente e um outro processo de servidor, que podem ser distinguidos um do outro, embora possam interagir totalmente;
b) a parte cliente e a parte servidor podem operar em diferentes plataformas de computador;
c) tanto a plataforma do cliente como a do servidor podem ser atualizadas independente uma da outra ;
d) o servidor pode atender a vários clientes simultaneamente. Em alguns sistemas cliente/servidor, os clientes podem acessar vários servidores;
e) os sistemas cliente/servidor incluem algum tipo de capacidade de operar em rede.

Os softwares localizados no cliente de um sistema cliente/servidor são normalmente chamados de softwares *front-end*. O cliente é o processo ativo na relação cliente/servidor sendo responsável pela inicialização e finalização das conversações com os servidores. Os clientes podem solicitar serviços distribuídos, mas não se comunicam diretamente com outros clientes, tornando a rede transparente ao usuário. Ferramentas de software *front-end* podem variar desde uma simples ferramenta de consulta para usuários comuns até ambientes de desenvolvimento de aplicativos cliente/servidor de missão crítica (BOCHENSKI, 1995, p. 26).

O sistema servidor, também conhecido como software *back-end*, possui uma execução contínua que recebe e responde requisições dos clientes, processando-as e devolvendo os resultados aos clientes. O servidor presta serviços distribuídos e atende diversos clientes simultaneamente, tendo como grande vantagem ser totalmente reativo, só disparando quando recebe alguma requisição do cliente. Isso faz com que o servidor não procure interagir com outros servidores durante um pedido de requisição, o que torna o processo de ativação uma tarefa a ser desempenhada apenas pelo cliente que o solicitou (DOERNER, 2004, p. 19).

2.5 TRABALHOS CORRELATOS

Entre os trabalhos correlatos no âmbito do VXt pode-se citar o trabalho da acadêmica Marilene Linzmeier que desenvolveu um tutorial para linguagem a*ssembly* utilizando o VXt (LINZMEIER, 1999). Outro projeto relacionado ao contexto do VXt foi proposto pelo acadêmico Marco Antonio Werka na forma de Trabalho de Conclusão de Curso, cujo objetivo

era a criação de um dispositivo CGA para o VXt (WERKA, 2001). Este trabalho foi iniciado, mas encontra-se suspenso neste momento.

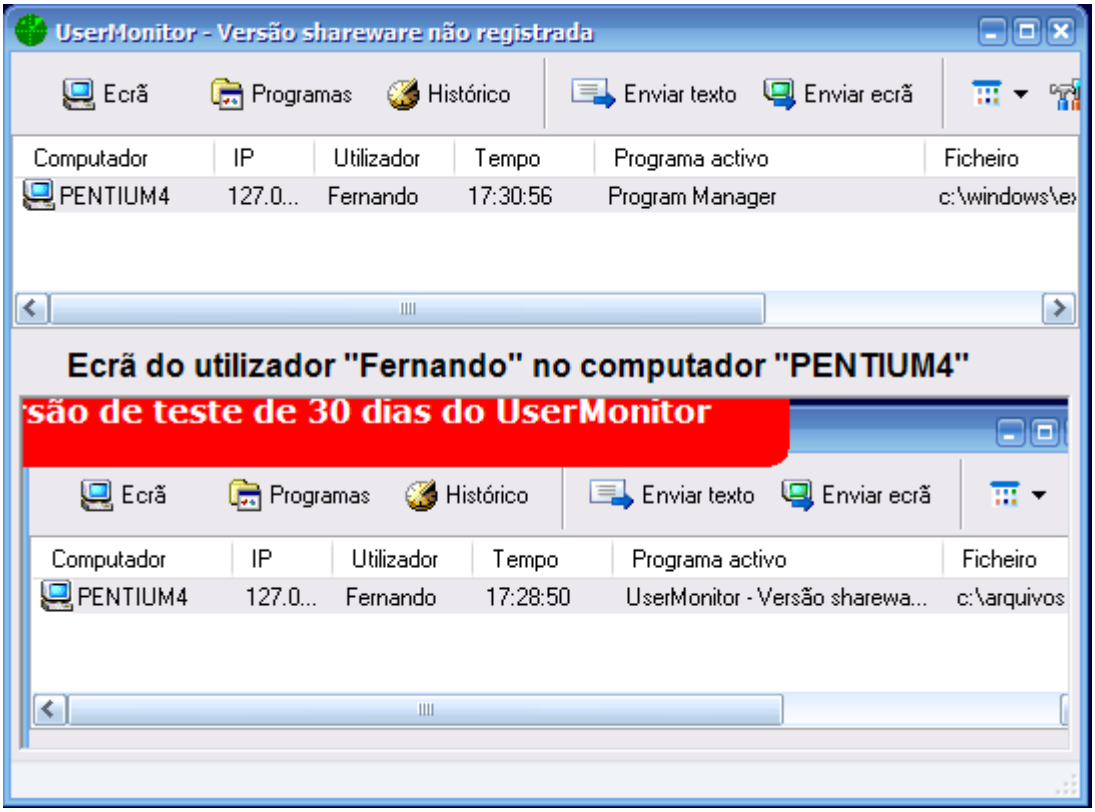
Além disso, foram desenvolvidos alguns projetos de pesquisa financiados pelo projeto PIPe/FURB envolvendo o VXt, os quais foram relatados em Mattos e Oliveira (1999b) que descreve o processo de integração das várias versões desenvolvidas do VXt em sala de aula; e mais recentemente Mattos et al. (2004) descrevendo a arquitetura mais recente do VXt que envolve a modelagem do sistema a partir de conceitos de Orientação a Objetos (OO).

Finalmente, em 2006 há a proposta de TCC do acadêmico Giordano B. S. Maciel (MACIEL, 2006) que pretende desenvolver um módulo *middleware* de hardware para permitir o desenvolvimento de *plugins* de hardware a serem anexados ao contexto do VXt.

Em termos de softwares educacionais na área de sistemas operacionais, existe o *Simulador para o Ensino de Sistemas Operacionais* (SOsim) que se encontra na versão 1.2. O SOsim permite que o professor apresente conceitos e mecanismos de um sistema operacional multiprogramável e/ou multitarefa, como Unix OpenVMS e Windows, tendo como características (MAIA, 2005):

a) mostrar o funcionamento de processos;
b) permitir a visualização de estruturas internas do sistema;
c) mostrar a gerência do processador e da memória.

No contexto de replicação de interface, há o software UserMonitor (USERMONITOR, 2005), que permite ao professor duplicar a interface do seu *desktop* para a tela dos alunos. Além disso, o software permite que o professor faça o monitoramento das atividades dos alunos consultando a interface de cada máquina individualmente a partir de seu computador. Com o UserMonitor também é possível que o professor receba mensagens de alertas quando os alunos executam programas não autorizados. A Figura 17 apresenta uma tela deste software. O software VirtualClass (VIRTUALCLASS, 2005) é outro exemplo de trabalho correlato que viabiliza a propagação da tela do professor para os alunos, permitindo também ao professor monitorar, inicializar e bloquear os programas que um aluno estiver utilizando em outro computador.



Fonte: Usermonitor, (2005).

Figura 17 – Tela do software UserMonitor

Relacionado ao modelo cliente/servidor existe o trabalho de Roberto César Tolardo (TOLARDO, 1996), que trata de uma biblioteca para comunicação entre aplicações cliente/servidor utilizando primitivas para desenvolvimento de aplicações do TCP/IP. O trabalho de Charles Thaler Trevizan (TREVIZAN, 1996), que desenvolveu um aplicativo cliente/servidor na implementação de um jogo de dominó multiusuario para ser jogado através da internet. O TCC de Joel Fernando P. de Farias (FARIAS, 1998), que consiste na construção de um protótipo do jogo de empresas LIDER, visando a sua operação em rede local de computadores através da arquitetura cliente/servidor. Este jogo é uma simulação computacional que pretende prever o comportamento humano dentro de uma realidade empresarial (FARIAS, 1998, p. 7). Também há o trabalho de Giovani Luebke (LUEBKE, 2003) que apresenta um protótipo de software que demonstra o aproveitamento de aplicações cliente/servidor para a internet usando o Forms da Oracle. E o trabalho de John Cristian Doerner (DOERNER, 2004) que demonstra o processo de desenvolvimento de um sistema de banco de dados relacional para ambiente cliente/servidor.

3 DESENVOLVIMENTO DO TRABALHO

Este capítulo contextualiza e descreve o desenvolvimento da extensão ao projeto VXt, apresentando os pontos de alteração necessários na versão em Delphi para permitir a interação com o *middleware* desenvolvido em Java. Descreve também a construção da interface cliente escrita em Java. Na elaboração e construção desta extensão ao projeto VXt foram realizadas as seguintes etapas:

a) elicitação dos requisitos: os requisitos funcionais e não-funcionais inicialmente identificados foram reavaliados e detalhados;
b) estudo da estrutura do VXt: a implementação da ferramenta VXt (MATTOS et al., 2004) foi analisada para identificar como o VXt foi concebido e qual a melhor estratégia a ser adotada para a viabilização da comunicação entre o VXt e o *middleware*;
c) especificação da interface cliente: foram especificadas as classes Java bem como a estrutura de classes a serem utilizadas na construção das réplicas dos formulários em Delphi usados no projeto VXt;
d) análise das informações apresentadas na interface em Delphi: foi analisado o volume de informações a serem propagados a partir da aplicação Delphi para as interfaces cliente a partir da análise das características de cada formulário Delphi;
e) especificação da ferramenta: foram especificados os diagramas de caso de uso e de classes utilizando os conceitos de orientação a objetos, através da *Unified Modeling Language* (UML);
f) implementação: para desenvolver as alterações no VXt foi utilizado o ambiente de desenvolvimento Delphi 7 enquanto que, para implementar o módulo *middleware* e as interfaces cliente, foi utilizado o ambiente Eclipse 3.2.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

A seguir são apresentados os requisitos funcionais e requisitos não funcionais, atendidos pelo software desenvolvido. O Quadro 23 apresenta os requisitos funcionais e sua rastreabilidade, ou seja, vinculação com o caso de uso associado. No Quadro 24 são

relacionados os requisitos não funcionais.

REQUISITOS FUNCIONAIS	CASO DE USO
RF01: o *middleware* deverá comunicar-se com o simulador do processador e mediar a propagação das informações de simulação entre os usuários conectados.	UC01
RF02: o *middleware* deverá gerenciar a entrada e saída de usuários ao longo do processo de simulação.	UC01
RF03: o *middleware* deverá permitir que um usuário administrativo obtenha o controle da interação com o simulador do processador.	UC01
RF04: o *middleware* poderá se comunicar com o cliente, através de uma janela de chat que será inicializada junto com ele.	UC01
RF05: o cliente Java poderá conectar-se ao servidor.	UC02
RF06: o cliente poderá salvar as instruções executadas para posterior análise.	UC02
RF07: o cliente poderá se comunicar com o *middleware* através de *chat*.	UC02

Quadro 23 – Requisitos funcionais

REQUISITOS NÃO FUNCIONAIS
RNF01: Manter o *layout* original dos componentes de visualização de dados.
RNF02: Utilizar a biblioteca *Swing* e a Máquina Virtual Java (JVM) 1.4 para a implementação do *layout* das telas cliente.
RNF03: confiabilidade na propagação do conteúdo da interface do VXt.
RNF04: o servidor de propagação e os clientes são escritos em Java.
RNF05: o *middleware* deverá ser de fácil utilização.
RNF06: o middleware desenvolvido em Java interage com o simulador da CPU do VXt escrito em Delphi.

Quadro 24 – Requisitos não funcionais

3.1.1 Técnicas e ferramentas utilizadas

A ferramenta aqui apresentada foi construída na linguagem Java, utilizando o ambiente de desenvolvimento Eclipse, o pacote Java 2 Software *Development* Kit (J2SDK) e a biblioteca gráfica *Swing* na versão 1.4.2. Para implementar as alterações no código do VXt foi utilizado o ambiente Delphi 7.0.

3.1.2 Implementação da ferramenta

A implementação da ferramenta iniciou com o estabelecimento do protocolo a ser utilizado para viabilizar a comunicação entre o VXt e o *middleware*.

Inicialmente adotou-se a seguinte notação conforme apresentado no Quadro 25.

Identificador	Finalidade
&campo01=	– Prefixo que identifica o campo da tela a ser atualizado
&<jan01>-open	– Sufixo associado a um identificador de tela <jan01> , indicando a janela que deve ser aberta nos módulos cliente
&<jan01>-close	– Sufixo, associado a um identificador de tela <jan01>, indicando a janela que deve ser fechada nos módulos cliente
<valor>	– Valor a ser atualizado em determinado campo de uma janela

Quadro 25 – Notação utilizada no protocolo de comunicação entre o VXt e o *middleware* e entre o *middleware* e os clientes de interface

3.1.3 Casos de Uso

Casos de uso referem-se aos serviços, tarefas ou funções que podem ser utilizados de alguma maneira pelos usuários do sistema (GUEDES, 2004, p. 46).

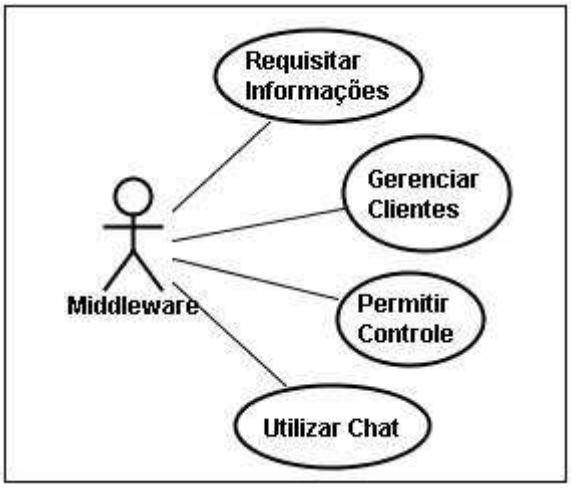


Figura 18 – Diagrama de caso de uso UC01 referente ao *middleware*

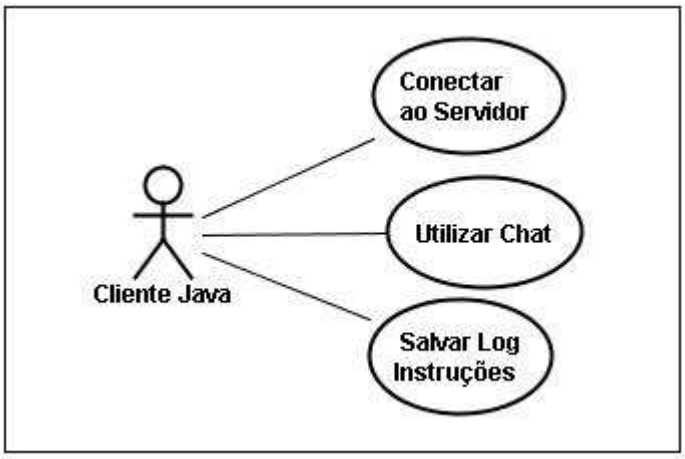


Figura 19 – Diagrama de caso UC02 referente ao cliente Java

3.2 DESCRIÇÃO DAS ALTERAÇÕES DO VXT

Para o desenvolvimento desse projeto, foi necessário implementar no VXt, uma comunicação entre ele e o *middleware* escrito em Java. Também tiveram de ser criadas novas *Unit,s* onde foram desenvolvidos os meta-forms, que são as *procedures* responsáveis por armazenar todas as alterações dos campos e dos componentes ocorridas na tela do VXt, para que possam ser transmitidas ao servidor e este repassar aos clientes em Java. Existem meta-forms para os formulários: principal, *log*, registradores e MDA (ver figura 20). Note que a idéia seria criar um meta-form para a janela CGA, porém a versão atual do VXt implementa somente uma interface MDA, visto que o TCC que deveria implementar uma controladora CGA não foi finalizado.

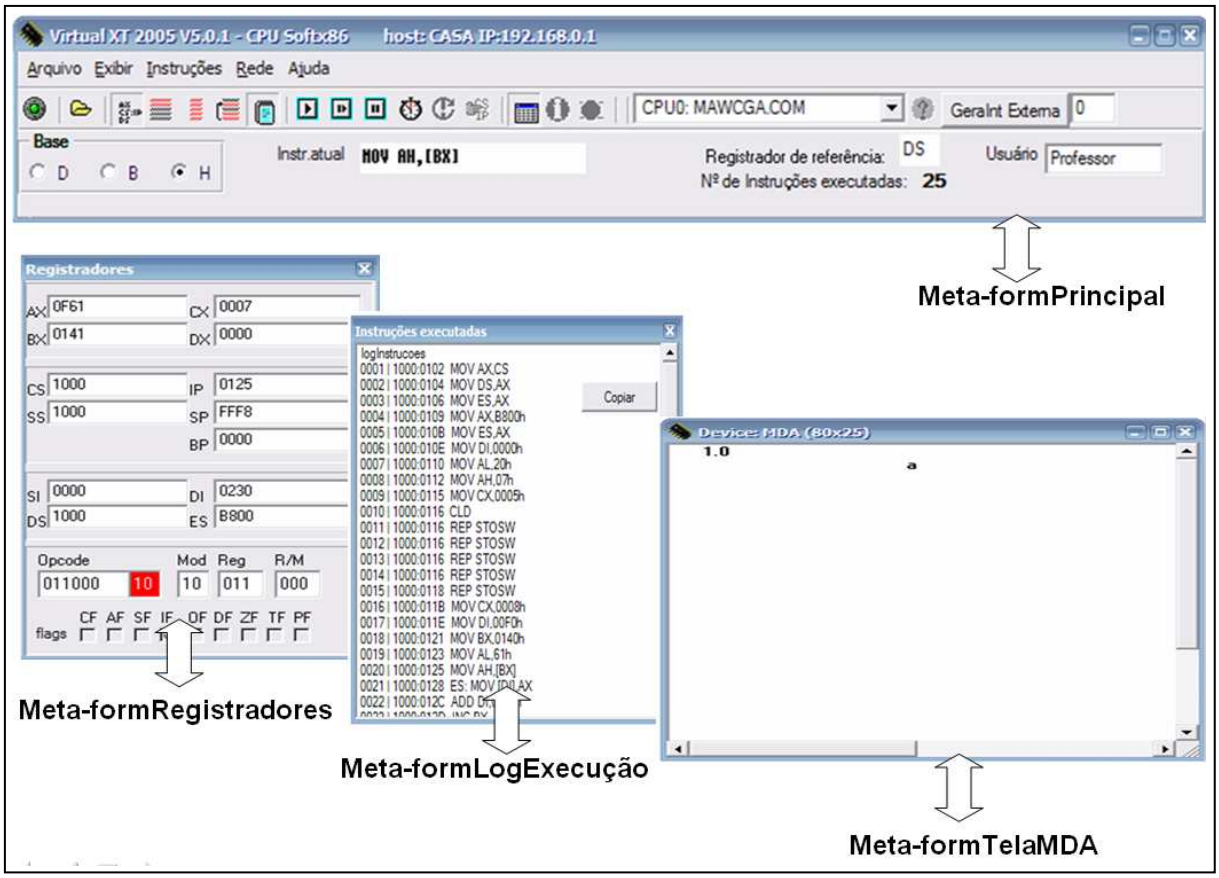


Figura 20 – Meta-forms das janelas do VXt

Esta estratégia reduziu significativamente o volume de tráfego na rede produzido entre o VXt em Delphi e os clientes em Java, já que não é necessário que sejam enviadas as informações de todos os campos existentes no sistema a cada comando de execução.

Quando uma instrução é executada no VXt, para todos os campos onde ocorreu uma alteração, é chamada uma *procedure*, referente aos meta-forms da janela alterada, para tratar desse evento. O Quadro 26 mostra o código que faz a chamada do procedimento pertencente a *Unit* MetaRegistradores para os campos AX e BX da janela de registradores.

```
procedure Tfrm_Registradores.Ed_AXChange(Sender: TObject);
begin
      metareg.recebeCampoEArmazena(Ed_AX);
end;

procedure Tfrm_Registradores.Ed_BXChange(Sender: TObject);
begin
      metareg.recebeCampoEArmazena(Ed_BX);
end;
```

Quadro 26 - Evento de atualização de um campo

Os valores dos campos alterados são adicionados num *buffer*, junto com o identificador

do seu campo e separados pelo símbolo “&” dos outros campos, como mostra o Quadro 27. Esse *buffer* será enviado junto com o das outras janelas do VXt, para o servidor, que o repassará para os clientes em Java.

```
procedure TMetaRegistradores.recebeCampoEArmazena(var cEdit:TEdit);
begin
         if cEdit.Name = 'Ed_Opcode' then
            regBuffer := regBuffer+'&Op1='+cEdit.Text
          else
          if cEdit.Name = 'Ed_Opcode2'then
             regBuffer := regBuffer+'&Op2='+cEdit.Text
          else
          if cEdit.Name = 'Ed_Mod' then
             regBuffer := regBuffer+'&Mod='+cEdit.Text
          else
          if cEdit.Name = 'Ed_Reg' then
                 regBuffer := regBuffer+'&Reg='+cEdit.Text
          else
              begin
                  nomeReg := Copy(cEdit.Name,4,5);
                  regBuffer:= regBuffer+'&'+nomeReg+'='+cEdit.Text;
              end;
end;//recebeCampoEArmazena
```

Quadro 27 – Código da *procedure* do Meta-formRegistradores

Com a criação desses meta-forms viabilizou-se a possibilidade de incluir novos formulários sem interferência na comunicação com o *middleware*.

O VXt conecta-se com o *middleware* utilizando *sockets*, baseando-se no modelo cliente/servidor. O fato do servidor ser escrito em outra linguagem (Java) não necessitou de nenhum tipo de procedimento especial para tratar essa situação. O Quadro 28 mostra a *procedure* que executa a conexão entre o cliente Delphi e o servidor em Java.

```
procedure
TClientFrmMain.conectaNoServidorJava(usuario:string;ipclient:string);
begin
   try
      Client.Connect(10000);
      Client.WriteLn(usuario+':conectado a partir do VXt Delphi
('+ipClient+')');
      ClientHandleThread := TClientHandleThread.Create(True);
      ClientHandleThread.FreeOnTerminate:=True;
      ClientHandleThread.Resume;
    except
      on E: Exception do MessageDlg ('Error na tentativa de
conexão.'+#13+'Verifique Servidor!', mtError, [mbOk], 0);
    end;
end;
```

Quadro 28 – *Procedure* que conecta cliente Delphi ao *middleware*

A cada comando de execução de uma instrução, é propagado para o *middleware*, o *buffer* contendo o valor de todos os campos onde houver uma atualização dos dados, realizados naquela interação, conforme demonstra o Quadro 29.

```
procedure
TClientFrmMain.enviaMensagemParaServidorJava(msg:string);
begin
  if (msg <> '') then
  begin
    if client.Connected then
      Client.Writeln(msg);
    IncomingMessages.Lines.Add(msg);
    EditMessage.Text:='';
  end;
end;
```

Quadro 29 – Código da *procedure* que envia dados para o *middleware*

3.3 ARQUITETURA DO *MIDDLEWARE*

Segundo Carvalho (2005), *middleware* é o neologismo criado para designar camadas de software que não constituem diretamente aplicações, mas que facilitam o uso de ambientes ricos em tecnologia da informação. A camada de *middleware* concentra serviços como identificação, autenticação, autorização, diretórios e outras ferramentas para segurança. Aplicações tradicionais implementam vários destes serviços, tratados de forma independente por cada uma delas. As aplicações modernas, no entanto, delegam e centralizam estes serviços na camada de *middleware*. Ou seja, o *middleware* serve como elemento que aglutina e dá coerência a um conjunto de aplicações e ambientes (CARVALHO, 2005).

A estratégia adotada para a concepção da arquitetura do *middleware* é apresentada na Figura 21.

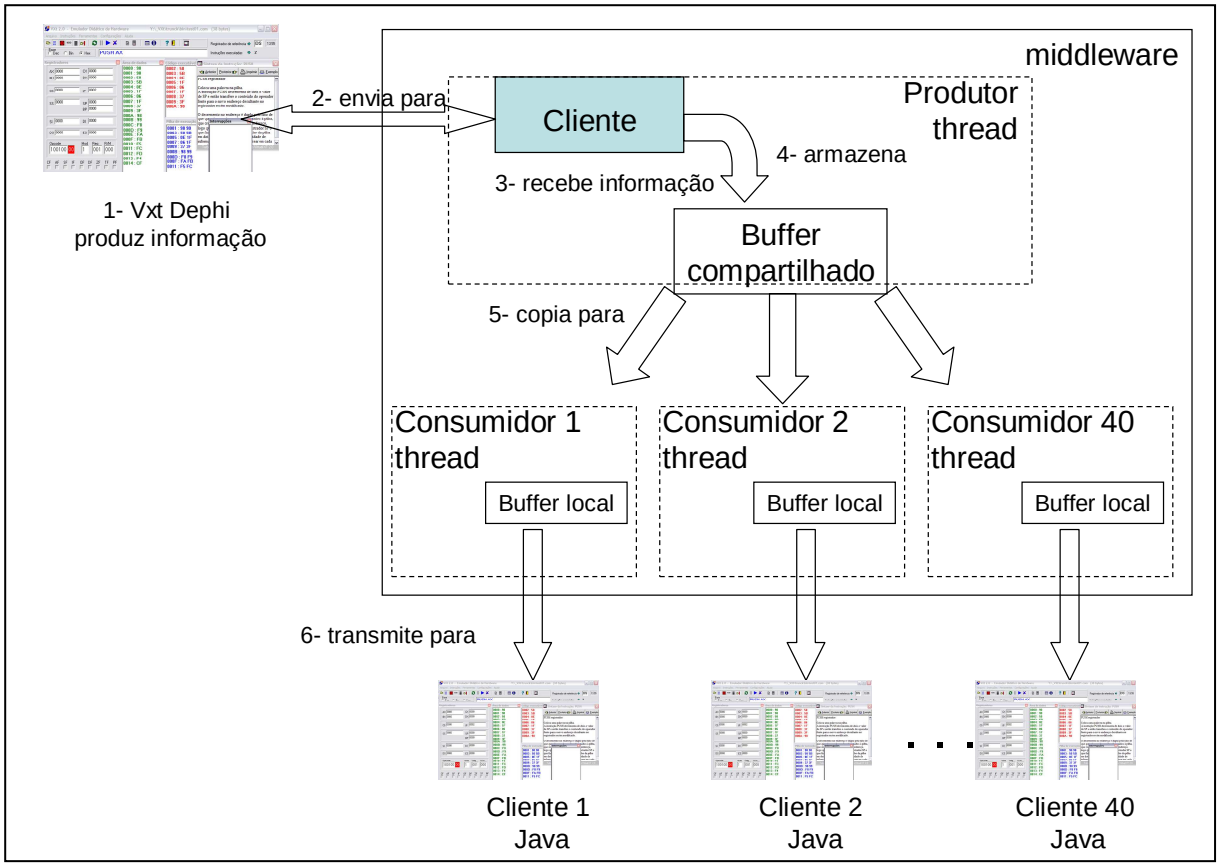


Figura 21 - Funcionamento da arquitetura produtor/consumidor do VXt

Como se pode observar, o *middleware* atua como um cliente que requisita informações que o VXt em Delphi (atuando como servidor) fornece.

Internamente, o *middleware* está estruturado na forma de um clássico modelo produtor/consumidor (CARISSIMI; OLIVEIRA; TOSCANI, 2003, p. 93). Ao receber uma mensagem do VXt, o servidor insere-a em um *buffer* global. Para cada cliente conectado é criada uma instância de um consumidor (*thread*) o qual, por sua vez, comunica-se exclusivamente com o seu cliente VXt Java.

A Figura 22 apresenta um diagrama de classes envolvendo os diversos componentes da camada *middleware*.

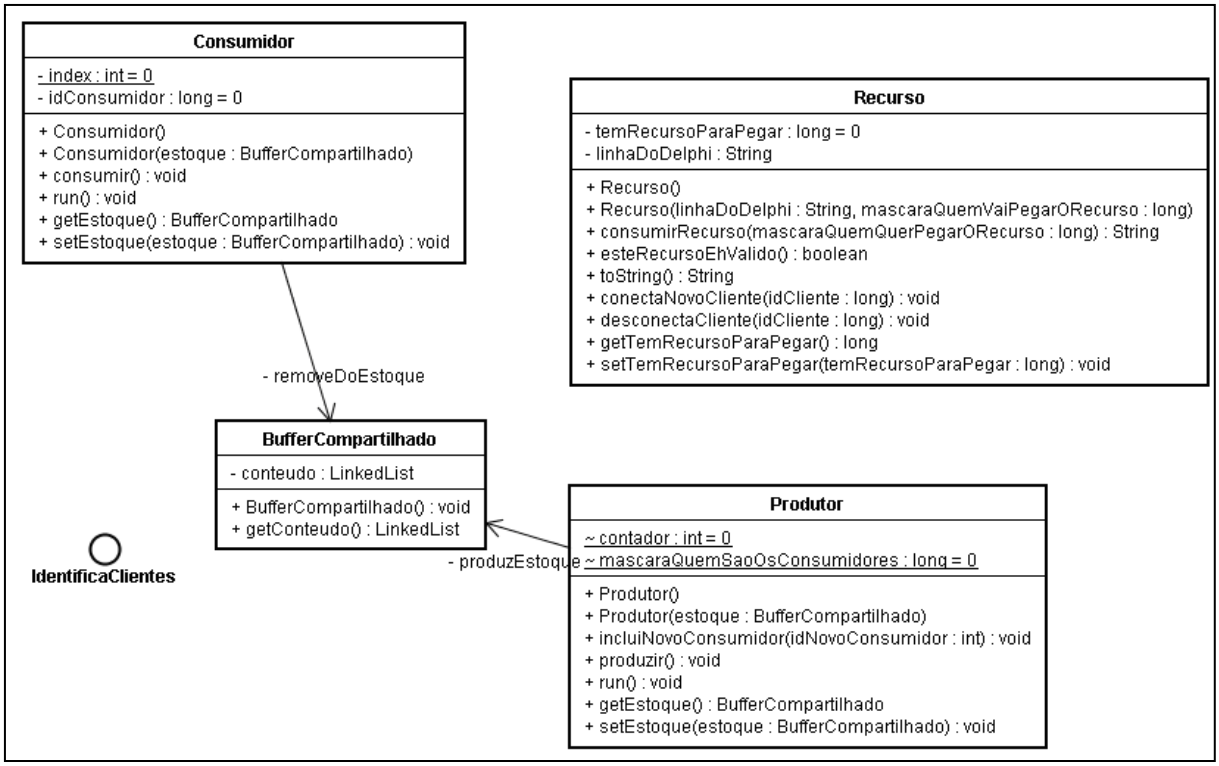


Figura 22 - Diagrama de classes do módulo *middleware*

A Figura 23 apresenta um diagrama de seqüência descrevendo a seqüência principal de passos que ocorre na comunicação entre o VXt em Delphi e o cliente VXt através do *middleware* em Java.

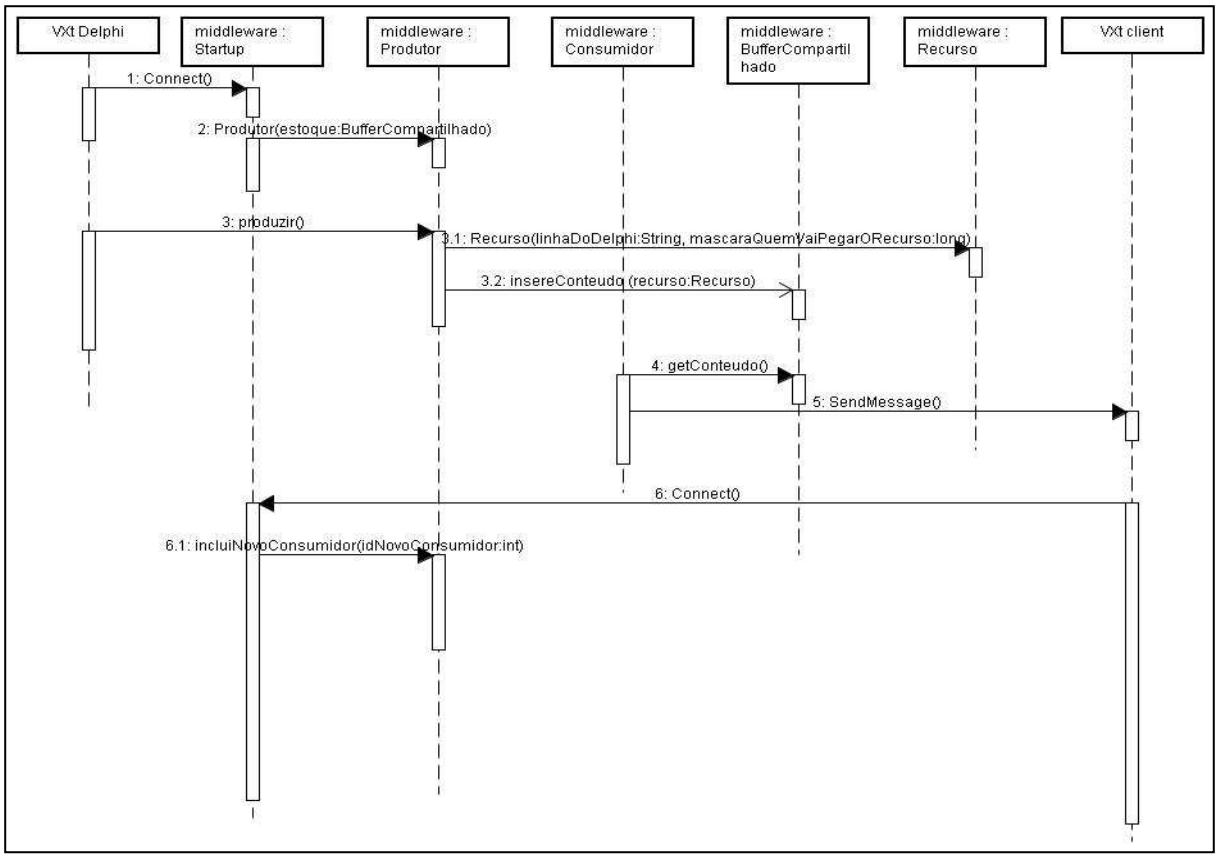


Figura 23 - Diagrama de seqüência

Já o Quadro 30 apresenta a declaração da interface que possui os identificadores dos clientes.

```
public interface IdentificaClientes {
    public static long codIdConsumidor[] = {
      1,                //    0000000000000000000000000000000000000001
      2,                //    0000000000000000000000000000000000000010
      4,                //    0000000000000000000000000000000000000100
      8,                //    0000000000000000000000000000000000001000
      16,               //    0000000000000000000000000000000000010000
      32,               //    0000000000000000000000000000000000100000
      64,               //    0000000000000000000000000000000001000000
      128,              //    0000000000000000000000000000000010000000
      256,              //    0000000000000000000000000000000100000000
      512,              //    0000000000000000000000000000001000000000
      1024,             //    0000000000000000000000000000010000000000
      2048,             //    0000000000000000000000000000100000000000
      4096,             //    0000000000000000000000000001000000000000
      8192,             //    0000000000000000000000000010000000000000
      16384,            //    0000000000000000000000000100000000000000
      32768,            //    0000000000000000000000001000000000000000
      65536,            //    0000000000000000000000010000000000000000
      131072,           //    0000000000000000000000100000000000000000
      262144,           //    0000000000000000000001000000000000000000
      524288,           //    0000000000000000000010000000000000000000
      1048576,          //    0000000000000000000100000000000000000000
      2097152,          //    0000000000000000001000000000000000000000
      4194304,          //    0000000000000000010000000000000000000000
      8388608,          //    0000000000000000100000000000000000000000
      16777216,         //    0000000000000001000000000000000000000000
      33554432,         //    0000000000000010000000000000000000000000
      67108864,         //    0000000000000100000000000000000000000000
      134217728,        //    0000000000001000000000000000000000000000
      268435456,        //    0000000000010000000000000000000000000000
      536870912,        //    0000000000100000000000000000000000000000
      1073741824,       //    0000000001000000000000000000000000000000
      2147483648l,      //    0000000010000000000000000000000000000000
      4294967296l,      //    0000000100000000000000000000000000000000
      8589934592l,      //    0000001000000000000000000000000000000000
      17179869184l,     //    0000010000000000000000000000000000000000
      34359738368l,     //    0000100000000000000000000000000000000000
      68719476736l,     //    0001000000000000000000000000000000000000
      137438953472l,    //    0010000000000000000000000000000000000000
      274877906944l,    //    0100000000000000000000000000000000000000
      549755813888l     //    1000000000000000000000000000000000000000
      };
}
```

Quadro 30 - Especificação da máscara de bits de identificação das *threads* consumidores.

A estratégia baseada em produtor/consumidor viabilizou que questões de comunicação específicas de algum cliente fiquem isoladas naquele cliente não comprometendo a execução dos demais.

Para operacionalizar esta estratégia, lançou-se mão de um atributo na classe do servidor chamado `mascaraQuemSaoOsConsumidores`. Ao ser incluído um novo consumidor através de uma conexão (tardia ou não) de algum cliente Java, o servidor seleciona um identificador de cliente (a partir da variável `idNovoConsumidor`) e recupera a máscara de bits que o identifica realizando uma operação OU com a mascara que identifica os indivíduos

atualmente conectados (Quadro 31).

```
public void incluiNovoConsumidor(int idNovoConsumidor) {
  mascaraQuemSaoOsConsumidores = mascaraQuemSaoOsConsumidores |
                                 codIdConsumidor[idNovoConsumidor];
        }
```

Quadro 31 – inclusão de um novo consumidor de recursos do *middleware*

Na Figura 24 é demonstrado um esquema de como é organizada a identificação dos consumidores, que são identificados pelo conjunto de valores presentes na interface `IdentificaClientes`.

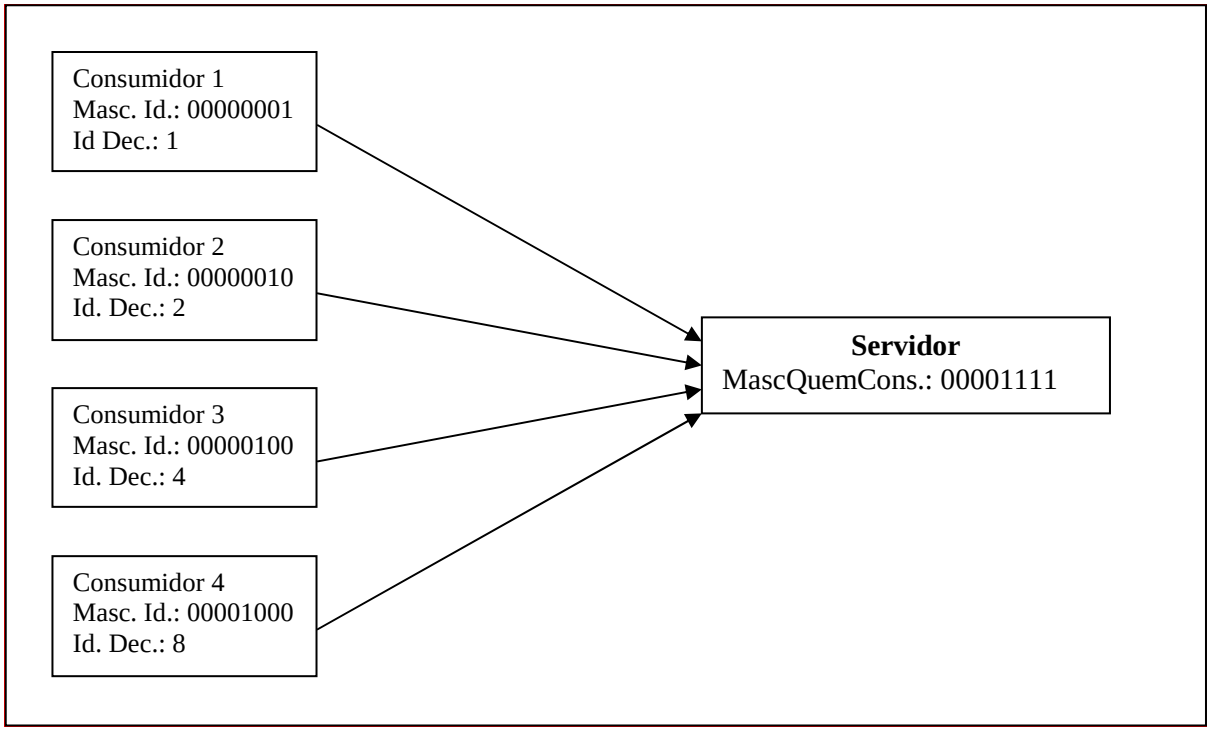


Figura 24 – Organização dos consumidores identificados

Quando um cliente vai consumir um recurso do *buffer* compartilhado, ele apresenta a sua máscara de identificação e, se já não consumiu aquele recurso que encontra-se no *buffer*, tem permissão de fazê-lo (Quadro 32).

```
public void consumir() {
  synchronized (estoque) {
    /* Verifica se existem itens no estoque */
   if (estoque.getConteudo().size() > 0) {
   //se entra aqui é porque tem estoque, ou seja, o delphi enviou uma
   //linha para o servidor. Minha funcao neste metodo é copiar do
   //arrayList do servidor para o arrayList deste consumidor.
   LinkedList<Recurso> listaDoServidor =
                           (LinkedList<Recurso>)estoque.getConteudo();
   Recurso r;
    for (int i=0; i < listaDoServidor.size();i++) {
     //vai buscar todas as mensagens para este consumidor
     r  = listaDoServidor.get(i);//pega um recurso da lista
     if (r != null) {
       if (r.esteRecursoEhValido()) {
         //Verifica se este recurso pode ser consumido por algum
         //Consumidor ou se já foi consumido por todos os consumidores
         //cadastrados. Se o flag temRecursoParaPegar == 0 significa que
         //todos os consumidores já consumiram o recurso
         String linhaDoDelphi = r.consumirRecurso(this.idConsumidor);
        }
        else {
          //se todos os Consumidores já consumiram este recurso, remove-o
         //da lista do produtor
          listaDoServidor.remove(i);
        }
     }
     else
        System.out.println(" nao tinha recurso "+this.getName());
    }//for
  }
  else {
   /* Não existe recursos no estoque */
   try {
       System.out.println("! " + this.getName() +  "\t -> Consumidor esperando
estoque ser reposto...");
      //Espera o produtor notificar que houve uma reposição no estoque
      System.out.println("consumidor "+this.getName()+ " vai aguardar");
      estoque.wait();
   }
   catch (InterruptedException e) {
            e.printStackTrace();
   }
                                          }
 }
}
```

Quadro 32 - Código do método consumidor

Inicialmente o método consumir verifica se existem itens no estoque, ou seja, se o VXt enviou uma informação para o servidor. A função deste método então, é de copiar do *buffer* compartilhado do servidor para o *buffer* da *thread* consumidora de cada cliente, que esteja executando o método consumir.

Para isto, executa-se um laço para permitir a este consumidor resgatar todos os recursos produzidos pelo VXt Delphi e copiá-los para o seu *buffer* local. Para isto, varre-se a estrutura de dados do *buffer* compartilhado, procurando por recursos que possuam na máscara de identificação um bit ligado correspondente ao identificador do consumidor atual. Se o *flag* `temRecursoParaPegar == 0`, significa que todos os consumidores já consumiram o recurso. Então, a função do último consumidor a varrer a lista, é remover aquele recurso do *buffer*

compartilhado.

Se não houver recurso em estoque, executa uma chamada `estoque.wait()` a qual bloqueia a *thread* até que o servidor informe que há recursos para consumir no *buffer* compartilhado.

Todo recurso ao ser criado recebe uma máscara de bits do servidor `(mascaraQuemVaiPegarORecurso)` identificando todos os clientes habilitados naquele momento a consumí-lo (Quadro 33).

```
public Recurso(String linhaDoDelphi,long mascaraQuemVaiPegarORecurso)
  this.temRecursoParaPegar = mascaraQuemVaiPegarORecurso;
  //cria um recurso armazenando uma linha recebida do delphi
  this.linhaDoDelphi = linhaDoDelphi;
}
```

Quadro 33 - Criação de um recurso com máscara de permissão de consumo.

Um aspecto a destacar é que, quando as *threads* compartilham acesso a um objeto comum, elas podem entrar em conflito umas com as outras (HORSTMANN, 2004, p. 777). Essa situação é denominada de condição de competição (ou *race condition*) (HORSTMANN, 2004, p. 779). Para solucionar o problema, cada *thread* deve ser capaz de bloquear um objeto temporariamente. Enquanto a *thread* bloqueia o objeto, nenhuma outra deve ser capaz de modificar o estado deste objeto. Na linguagem de programação Java utilizam-se métodos sincronizados para bloquear objetos (HORSTMANN, 2004, p. 784) através da palavra-chave `synchronized` (Quadro 32).

Quando uma *thread* consumiu todos os seus recursos e não tem mais o que fazer, ela chama o método *wait*() (Quadro 32) liberando o processador para que outras *threads* executem.

Quando um novo recurso é adicionado ao *buffer* compartilhado, o método do servidor chama `notifyAll` para desbloquear todas as *threads* consumidoras para que elas resgatem o recurso do *buffer* global e transmitam para seus respectivos clientes VXt.

3.4 MÓDULO CLIENTE

A interface do cliente Java é idêntica ao do VXt em Delphi, porém a sua única ação, é conectar-se ao servidor. O usuário, em sua interface, somente será capaz de acompanhar a execução do VXt que estará sendo rodado numa outra máquina junto com o servidor.

A Figura 25 apresenta o diagrama de classes do módulo cliente Java do VXt.

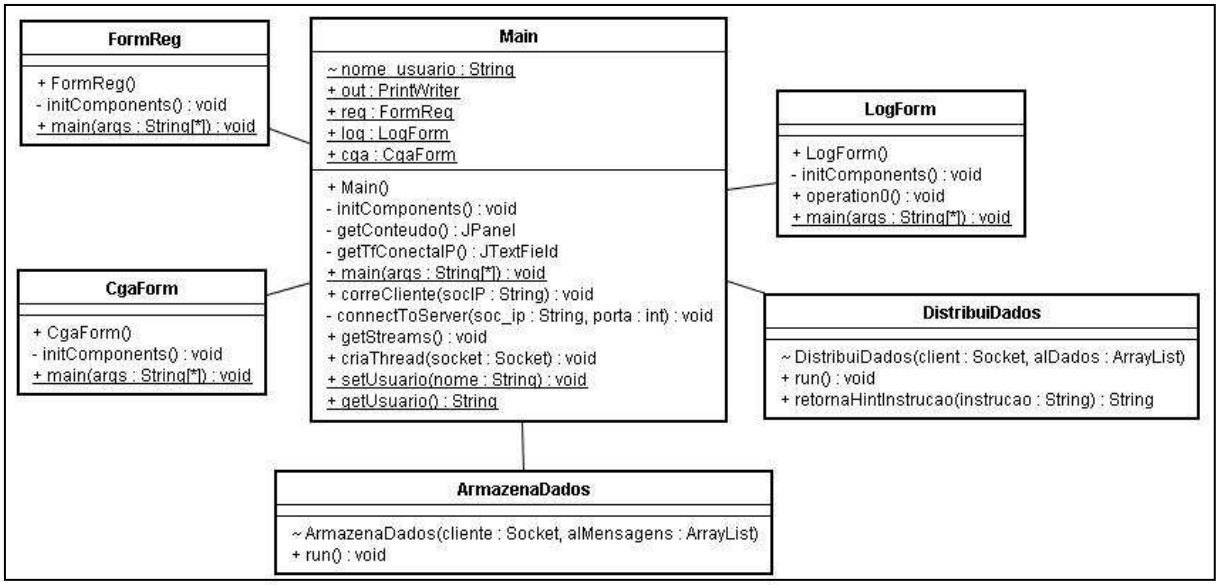


Figura 25 – Diagrama de classes do cliente Java

Toda vez que o VXt aciona a abertura ou fechamento de uma janela, a mesma ação ocorrerá com o cliente, que abrirá ou fechará a janela correspondente em sua interface.

As informações que o cliente Java recebe do *middleware*, caso não sejam responsáveis por abrir ou fechar alguma janela, são primeiramente armazenadas em um *ArrayList*, a partir do qual são recuperadas as informações na ordem em que chegaram, atualizando os campos e as janelas da sua interface, sendo posteriormente removida para a utilização da próxima informação que se encontra na fila de dados do *array*. Esse procedimento é necessário para que não se perca nenhuma informação proveniente do *middleware*, numa situação em que o servidor transmita mais rapidamente do que a capacidade do cliente de processar as mensagens recebidas.

A conexão entre o cliente Java e o *middleware* acontece através do uso de *sockets*. O cliente executa o método `connectToServer` onde se passa como parâmetro o IP e a porta de onde se encontra o servidor.(Quadro 34).

```
/**Executa conexão com servidor */
public  void correCliente(String socIP) {

      connectToServer(socIP, 7000);
      if (socThread == null){ // Trata erro de conexão com servidor
                  JOptionPane.showMessageDialog(null,"Servidor não esta em
                  execução!","Aviso", JOptionPane.INFORMATION_MESSAGE);
      }else{
            System.out.println("Servidor conectado");
            conectado = 1;
            getStreams();

            btConectar.setEnabled(false);
            miChatServ.setEnabled(true);
            setUsuario(tfAcesso.getText());

            jMDesConect.setEnabled(true);
             tfAcesso.setEditable(false);
             jMConecta.setEnabled(false);
             lbAtivo.setText("Conectado");
            lbAtivo.setForeground(new Color(0,0,255));

            criaThread(socThread);
            out.println("***"+ getUsuario());    // nome do usuario
             this.setTitle("Virtual XT 2006 - Java Cliente - "+
getUsuario());
            return;
      }
}
```

Quadro 34 – Conexão do cliente com middleware

3.5 *CHAT*

Essa versão do VXt, possibilita a comunicação entre os clientes (alunos) e o servidor (professor) através da utilização do recurso de *chat* baseado no modelo cliente/servidor. O servidor poderá mandar mensagens para somente um cliente em específico, ou para todos os clientes que estejam conectados de uma só vez (Figura 26). O cliente somente poderá se comunicar com o servidor e com nenhum outro cliente que esteja conectado (Figura 27).

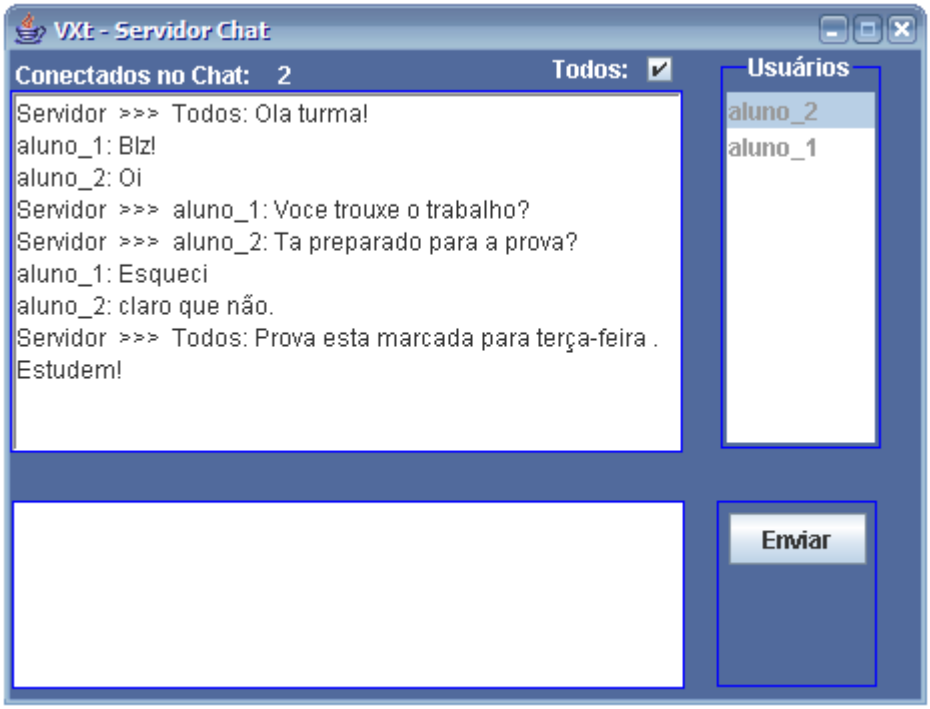


Figura 26 – Tela de *chat* do *middleware*

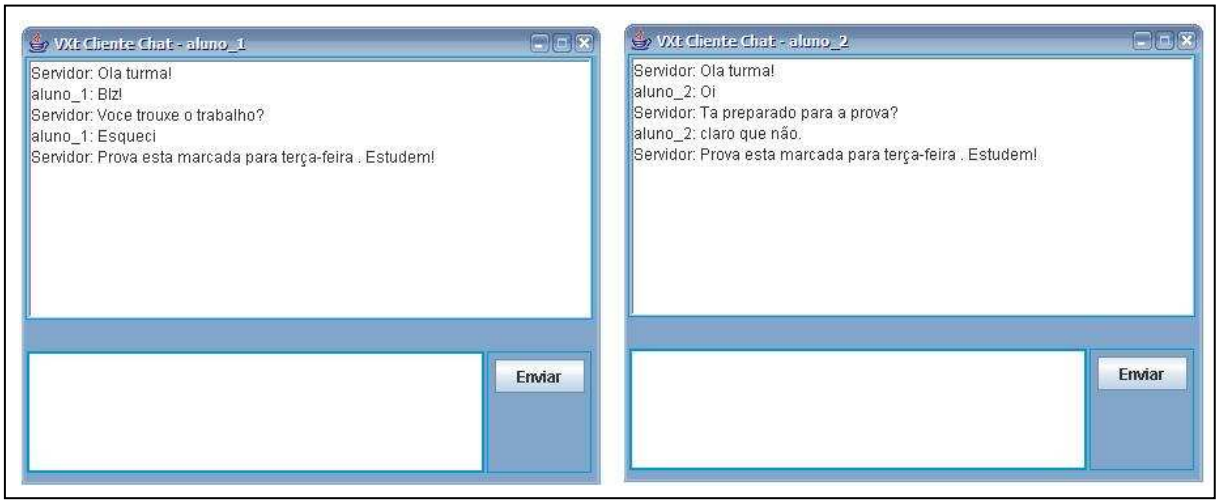


Figura 27 – Tela de *chat* dos clientes

3.6 SEQÜÊNCIA DE FUNCIONAMENTO DO VXT

Para a execução dessa versão do VXt, primeiramente deve-se executar o servidor, para posteriormente ser conectado a ele o VXt em Delphi e os clientes em Java.

Depois de conectado, o VXt abre o arquivo no qual irá simular os eventos do processador, executa as ações de simular, transmitindo todas as informações de seus campos

e janelas que foram atualizados, para o *middleware* (Figura 28).

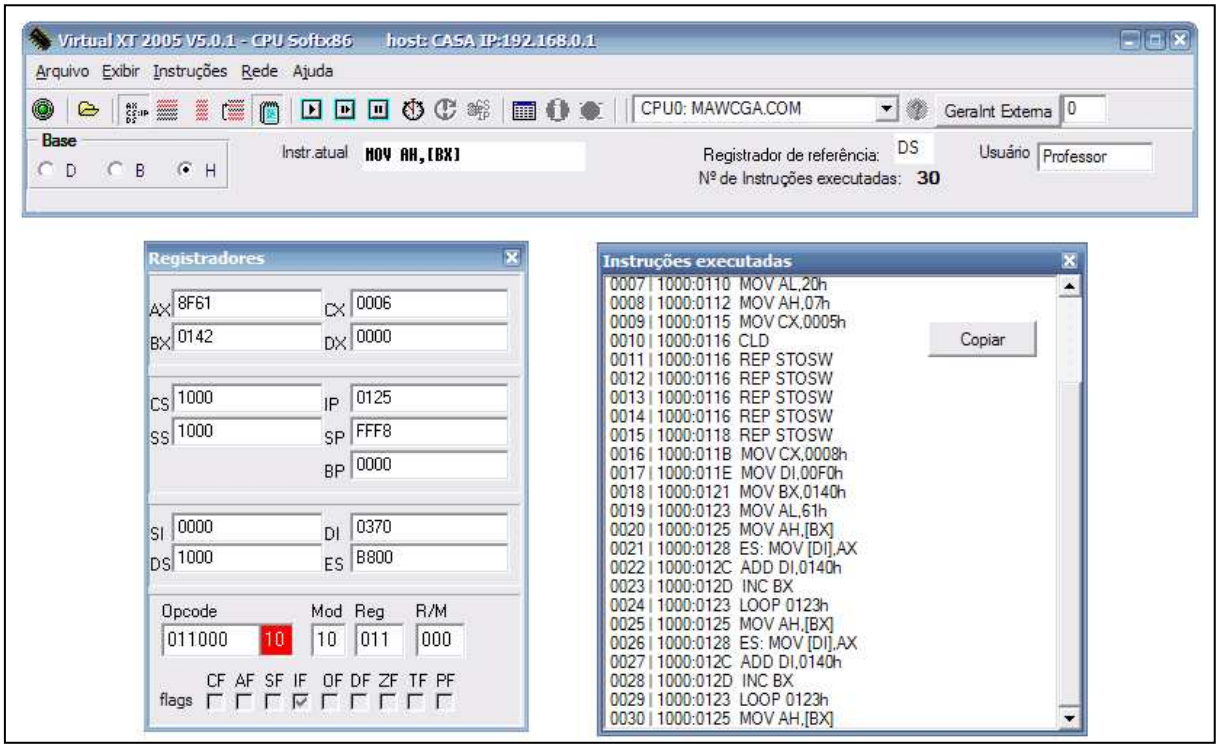


Figura 28 – Tela principal do VXt com janela registradores e *log* de instrução abertas

O Quadro 35 demonstra o trecho de código do arquivo *mawcga.com* utilizado para a execução do VXt. Este trecho é responsável por escrever a letra “a” na tela da janela MDA.

```
;part 2: write the character 'a' with 8 different attributes down the
center of the screen
;        with a blank line in between each character
;
      mov     cx,8      ;loop counter for 8 attribute bytes
      mov     di,240    ;initial offset into display memory for center
of second line
      mov     bx,offset attributes     ;pointer to the attribute bytes
      mov     al,'a'     ;character to be displayed
;loop to display 'a' with b attributes
display: mov  ah,[bx]    ;get new attribute byte
         mov  es:[di],ax ;move charactercode and attribute byte to
adapter memory
            add     di,320    ;point to display position tso lines down
            inc     bx        ;point to nest attribute byte
            loop    display   ;do it for 8 attribute bytes
```

Quadro 35 – Trecho do código do arquivo executado no VXt

A cada novo cliente que se conecta ao servidor, é adicionado o seu nome de usuário na área de clientes do *middleware* e também é incrementada a quantidade de conectados, que se encontra disponível no título de seu frame (Figura 29). Deve-se notar que o cliente VXt em Delphi não faz parte dessa lista.

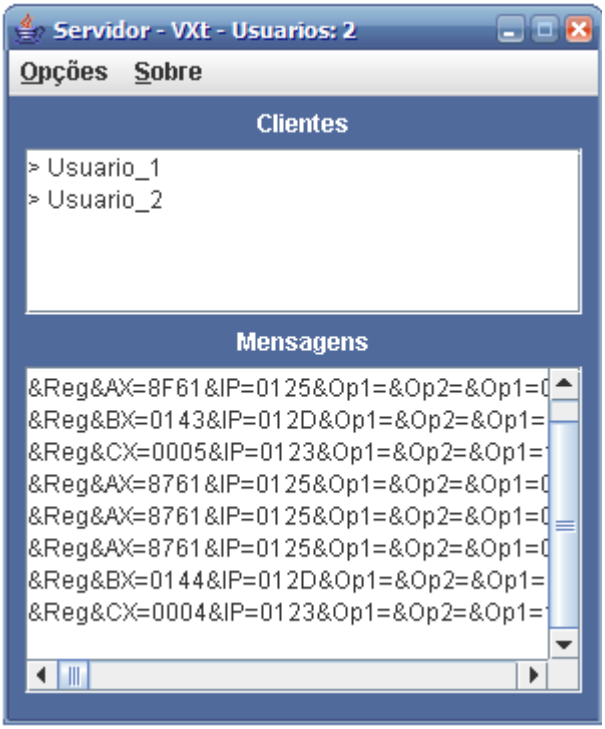


Figura 29 – Servidor VXt

A área de mensagens do servidor, mostra o formato das mensagens que ele recebe do VXt. A mensagem recebida contem um *buffer* com todos os campos que sofreram alguma modificação junto com seu identificador e seu valor, separados pelo símbolo “&” (Figura 29).

Os clientes Java recebem esses *buffers* e fazem uma separação das mensagens baseando-se no código do campo e no símbolo “&”. Depois de separados, os valores substituem os seus devidos campos nas janelas das interfaces dos clientes (Figura 30).

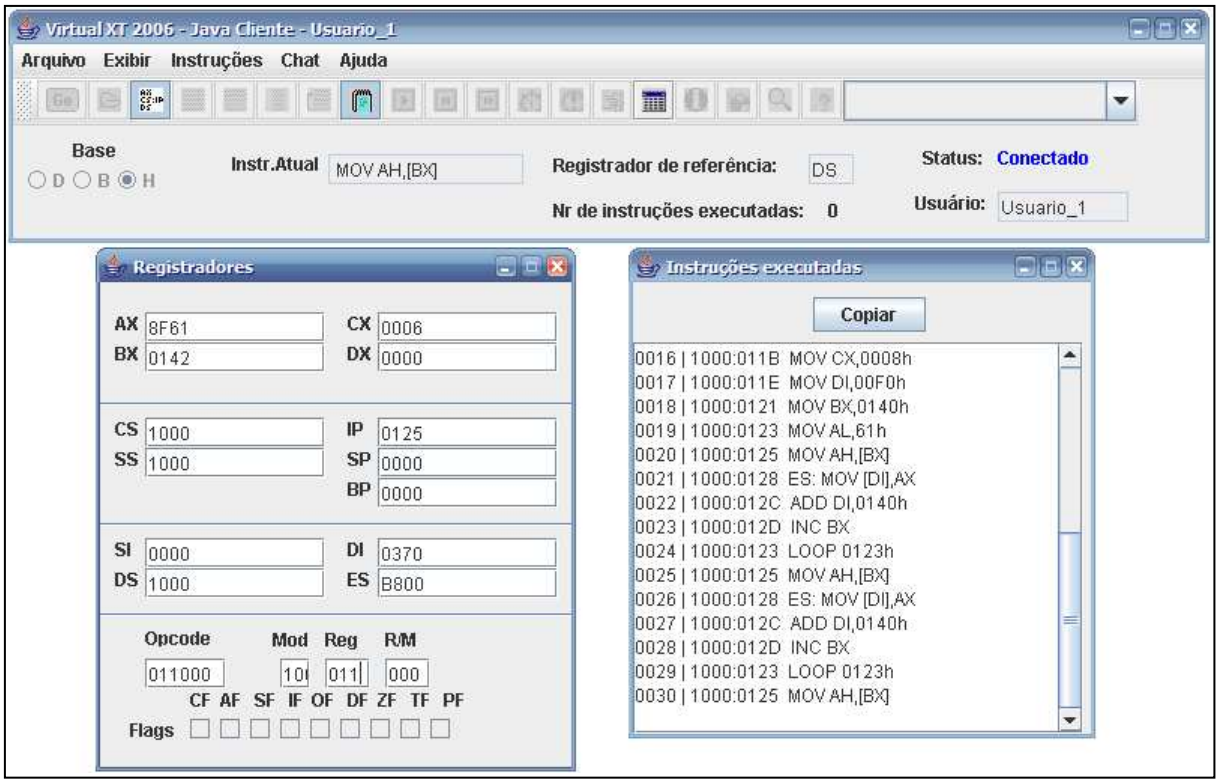


Figura 30 – Interface VXt cliente Java com registradores e *log* de instrução

3.6.1 Operacionalidade da implementação

Nesta seção é apresentada a operacionalidade da ferramenta. A versão VXt cliente/servidor implementada é apresentada na Figura 31 sendo executada com 2 usuários numa mesma máquina.

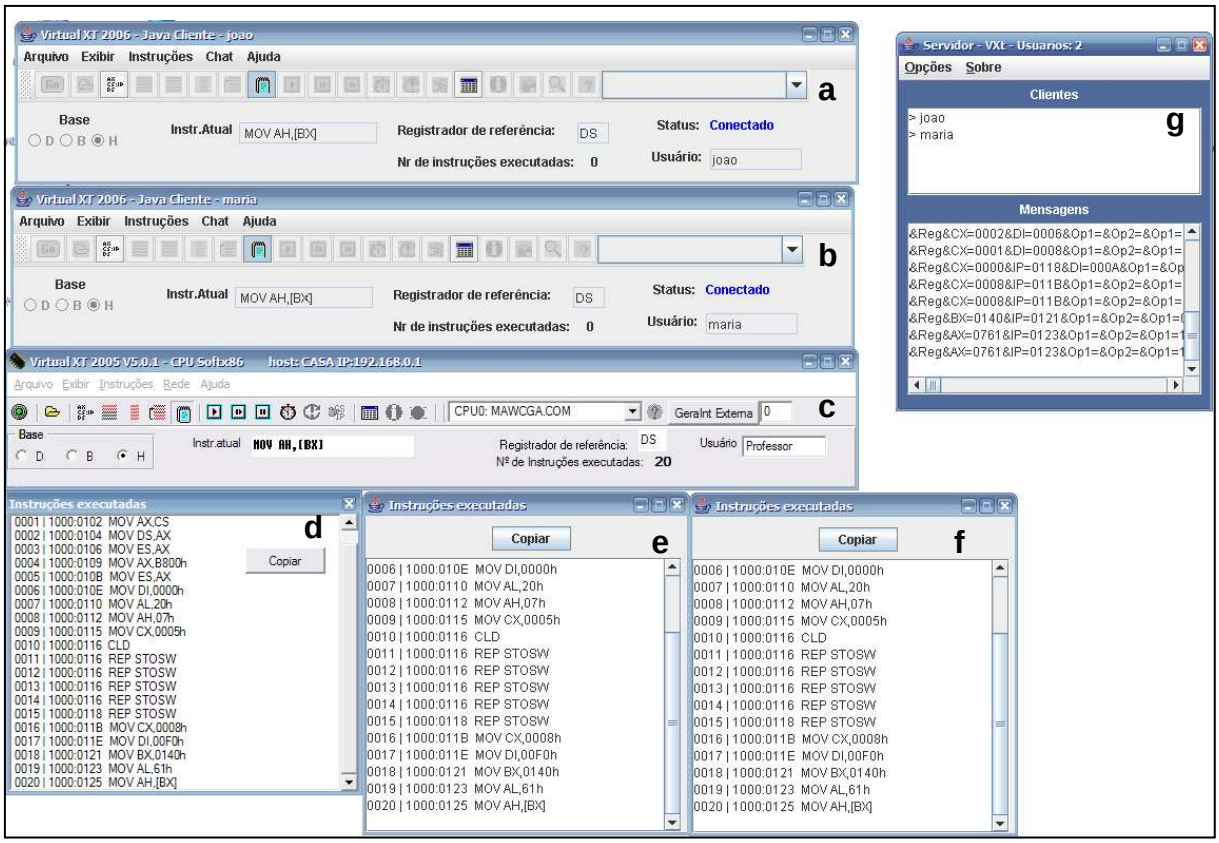


Figura 31 – Funcionamento do sistema em somente um computador

Pode-se observar que o conteúdo da tela do cliente Delphi é replicado para todas as interfaces Java através do *middleware* em execução. A seguir é demonstrada a descrição de cada janela presente na figura 31, sendo estas rotuladas com as letras do alfabeto:

a) cliente Java 1: recebe dados do *middleware*;
b) cliente Java 2: recebe dados do *middleware*;
c) cliente Delphi: envia dados para o *middleware*;
d) janela de *l*og de instruções do cliente Delphi;
e) janela de *log* de instruções do cliente Java 1;
f) janela de *log* de instruções do cliente Java 2;
g) *middleware* Java: gerencia clientes, recebe mensagens do VXt e trasmite para as interfaces Java.

3.7 RESULTADOS E DISCUSSÃO

Como se pode observar, a versão implementada atende aos requisitos propostos, sendo

possível propagar a interface do VXt para clientes remotos. Com essas novas características implantadas no VXt, ele torna mais interessante a aprendizagem dos alunos, pois estes, poderão ver na prática o funcionamento de uma CPU, assimilando melhor os assuntos relacionados a sistemas operacionais.

Com relação a performance da transmissão dos dados, pôde-se constatar que o tempo de atraso desde o envio até o recebimento pelos clientes das informações, não foi significativo a ponto de comprometer a aplicação.

Com essa versão foi possível propagar a interface do VXt sem perdas de informações. Num teste realizado com 20 computadores conectados ao *middleware*, praticamente não houve tempo de atraso no envio de informações a 500 ms.

Através de testes realizados com o VXt, foi observado que ele também funciona utilizando como meio de transmissão a internet, sendo o teste realizado somente entre 2 computadores.

Também foram feitos testes com os clientes sendo executados no sistema operacional Linux e notou-se que o VXt propaga sua interface sem nenhum problema para esta plataforma.

Os clientes não conseguem obter o controle do VXt conforme foi especificado no trabalho, mas, pelo fato de poderem tanto enviar (através do *chat*) como receber mensagens do *middleware*, foi dado um passo inicial relevante, para a conclusão desse objetivo.

4 CONCLUSÕES

O presente trabalho retrata o projeto de uma ferramenta em Java que, fazendo uso do modelo cliente/servidor, possibilita a propagação da interface de execução do VXt para as máquinas de uma sala de aula. Para tanto, foram descritos os requisitos do software a ser desenvolvido e apresentado os principais conceitos relacionados ao contexto do projeto. Foram apresentados também trabalhos correlatos desenvolvidos no projeto VXt , relacionados ao contexto de propagação de interface com o usuário e um trabalho relacionado ao uso de ferramenta de simulação para apoio ao ensino de sistemas operacionais.

Conforme apresentado anteriormente, a propagação da interface para os alunos de uma sala de aula permite ampliar as formas de utilização da ferramenta contribuindo desta forma para a melhoria da qualidade das aulas melhorando o entendimento dos alunos nos assuntos relacionados ao funcionamento de uma CPU.

A ferramenta também apresentou como solução a utilização do padrão de projeto MVC, facilitando a organização das classes geradas e visando melhorar a manipulação do código de saída.

A utilização de múltiplas *threads* foi muito importante para que fosse possível, uma eficiente transmissão da interface do VXt, sendo um item que tomou grande parte do tempo destinado ao desenvolvimento do trabalho. A partir do momento que se utilizou o modelo produtor/consumidor como forma de envio e recebimento de dados, eliminou-se a maior parte dos problemas que barravam o avanço do trabalho, que até então, se encontrava progredindo de forma muito lenta.

Também observou-se, que a sincronização de múltiplas *threads*, quando utilizadas de forma correta, torna eficiente, segura e simples as transmissão de várias informações aos mesmo tempo, ocorridas entre o servidor e os vários clientes conectados.

4.1 EXTENSÕES

Como extensões sugere-se:

a) permitir que o cliente Java assuma o controle da aplicação Delphi;

b) incrementar o *middleware* com a possibilidade de integrar dispositivos de

hardware simulado;

c) acrescentar novas janelas para serem observadas pelos clientes conectados;
d) transmissão da posição da seta do mouse para que os alunos saibam para onde o professor estaria apontando no VXt.

REFERÊNCIAS BIBLIOGRÁFICAS

BOCHENSKI, Barbara. **Implementando sistemas cliente/servidor de qualidade**. São Paulo: Makron Books, 1995.

BRAZ, Christian C. M. **Principais padrões J2EE para a construção de aplicações não distribuídas**. Rio de Janeiro, 2006. Disponível em: <http://www.devmedia.com.br/visualizacomponente.aspx?comp=1812&site=6> Acesso em: 03 nov. 2006.

CARISSIMI Alexandre S.; OLIVEIRA Rômulo S.; TOSCANI, Simão S. **Sistemas operacionais e programação concorrente**. São Paulo: Sagra Luzzatto, 2003.

CARVALHO, Osvaldo. **GT Middleware.** [S.l.], 2005**.** Disponível em: <http://www.rnp.br/pd/gts2004-2005/middleware.html>. Acesso em: 28 ago. 2006.

CASAS, Luis A. A. **Contribuições para a modelagem de um ambiente inteligente de educação baseado em realidade virtual**. 1999. Tese (Doutorado em Engenharia de Produção) - Departamento de Engenharia de Produção, Universidade Federal de Santa Catarina, Florianópolis. Não paginado. Disponível em: <http://www.eps.ufsc.br/teses99/casas/index.html>. Acesso em: 24 maio 2006.

DOERNER, John C. **Protótipo de um banco de dados relacional cliente/servidor**. 2004. 87 f. Trabalho de Conclusão de Curso (Bacharelado de Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

FARIAS, Joel F. P. **Jogo de empresas Líder em ambiente cliente/servidor**. 1998. 55 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

FUNDÃO DA COMPUTAÇÃO. **Design patterns fundamentais do J2EE**. [S.l.], 2004. Disponível em: <http://www.linhadecodigo.com.br/artigos.asp?id_ac=363>. Acesso em: 12 out. 2006.

GAMMA, E. et al. **Design patterns**: elements of reusable object-oriented software. Massachusetts: Addison-Wesley, 1994.

GUEDES, Gilleanes T. A. **UML**: uma abordagem prática. São Paulo: Novatec, 2004.

HANSEN , K. H. **StrutsTestCase**: the tool for struts unit testing. [S.l.], [2006?]. Disponível em: <http://javaboutique.internet.com/tutorials/StrutsTestCase>. Acesso em: 30 out. 2006.

HORSTMANN, Cay. **Big Java**. São Paulo: Bookman, 2004.

LINZMEIER, Marilene. **Tutorial da linguagem Assembly utilizando o VXt**. 1999. 58 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

LUEBKE, Giovani. **Protótipo de software para demonstrar o aproveitamento de aplicações cliente-servidor para ambiente web utilizando o Application Web Server da Oracle**. 2003. 51 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

MACIEL, Giordano B. S. **Implementação de um middleware para plug-in de dispositivos de hardware simulados para o VXt**. 2006. 18 f. Proposta de Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

MAIA, Luiz P. **SOsim:** simulador para o ensino de sistemas operacionais versão 1.2. [S.l.], 2005. Disponível em: < http://www.training.com.br/sosim/>. Acesso em: 10 mar. 2006.

MATTOS, Mauro M. **Fundamentos Conceituais para a Construção de Sistemas Operacionais Baseados em Conhecimento**. 2003. 382 f. Tese (Doutorado em Engenharia de Produção) – Curso de Pós-Graduação em Engenharia de Produção da Universidade Federal de Santa Catarina, Florianópolis.

MATTOS, Mauro M. et al. VXt**:** um ambiente didático para ensino de conceitos básicos de sistemas operacionais e arquitetura de computadores. In: WORKSHOP DE COMPUTAÇÃO DA REGIÃO SUL, 1., 2004, Florianópolis. **Anais…** Florianópolis: Unisul, 2004. Paginação irregular.

MATTOS, Mauro M.; OLIVEIRA, Emerson. Desenvolvimento de um ambiente de apoio ao ensino de conceitos básicos de hardware e software. In: SEMINÁRIO INTEGRADO DE INICIAÇÃO CIENTÍFICA, 5., 1999, Joaçaba. **Anais...** Joaçaba: FURB-UNIVALI, 1999. p. 79.

MATTOS, Mauro M.; TAVARES, Antonio C. Virtual XT: um ambiente de apoio ao ensino de conceitos básicos de hardware e software. In: SEMINARIO DA COMPUTAÇÃO, 6., 1997, Blumenau. **Anais...** Blumenau: FURB, 1997 p. 70-74.

MATTOS, Mauro M.; TAVARES, Antonio C. Desenvolvimento cooperativo de um ambiente de apoio ao ensino de conceitos básicos de hardware e software. In: WORKSHOP DE ENSINO EM INFORMÁTICA, 7., 1999, Rio de Janeiro. **Anais...** Rio de Janeiro: [S.l], 1999a. Paginação irregular.

MATTOS, Mauro. M. ; TAVARES, Antonio C. VXt: experiência de desenvolvimento cooperativo de um ambiente didático. In: CONGRESO IBEROAMERICANO DE EDUCACIÓN SUPERIOR EN COMPUTACION, 7., 1999, Asunción. **Anais…** Asunción: [S.l.], 1999b. Paginação irregular.

MATTOS, Mauro M.; TAVARES, Antonio C.; OLIVEIRA, Emerson. VXt: descrição da implementação de um simulador de hardware. In: SEMINÁRIO DA COMPUTAÇÃO, 7., 1998, Blumenau. **Anais…** Blumenau: FURB, 1998. p. 138-149.

METSKER, Steven J. **Padrões de projeto em Java**. Porto Alegre: Bookman, 2004.

PADRÕES de projeto de software. In: WIKIPEDIA, a enciclopédia livre. [S.l.]: Wikimedia Foundation, 2006. Disponível em: <http://pt.wikipedia.org/wiki/Padr%C3%B5es_de_projeto_de_software>. Acesso em: 12 out. 2006.

SILVEIRA, Janira. **Extensão da ferramenta Delphi2Java-II para suportar componentes de bancos de dados**. 2006. 81 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

SOURCEFORGE. **Software x86 CPU emulator library**: help. Version 0.00.0033. [S.l.], 2003. Documento eletrônico disponibilizado com o Softx86. Disponível em: <http://sourceforge.net/search/?type_of_search=soft&words=softx86>. Acesso em: 02 nov. 2006.

TOLARDO, Roberto C. **Uma biblioteca de funções para comunicação entre aplicações cliente-servidor utilizando o protocolo TCP da arquitetura de rede TCP/IP.** 1996. 55 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

TREVIZAN, Charles T. **Desenvolvimento de um jogo multiusuário através de plataforma cliente-servidor utilizando TCP/IP**. 1996. 47 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

USERMONITOR. **UserMonitor**. Halle, 2005. Disponível em: <http://www.neuber.com/usermonitor/index.html>. Acesso em: 8 abr. 2006.

VEIGA, Marise S. Computador e educacão? Uma ótima combinação. In: BELLO, José L. P. **Pedagogia em foco**. Petrópolis, 2001. Disponível em: <http://www.pedagogiaemfoco.pro.br/inedu01.htm>. Acesso em: 14 jun. 2006.

VIRTUAL CLASS. **VirtualClass™ 4.0**. Irvine, 2005. Disponível em: <http://www.farstone.com/home/ensite/products/virtualclass.shtml>. Acesso em: 11 jul. 2006.

WERKA, Marco A. **Proposta de uma interface CGA para o projeto XT**. 2001. 11 f. Proposta de Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.