UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

“IMPLEMENTAÇÃO DOS SERVIÇOS DE COMUNICAÇÃO E MONITORAÇÃO EM UM AMBIENTE PARA DESENVOLVIMENTO DE SISTEMAS DISTRIBUIDOS”

ORIENTADOR: Prof. Dr. Célio Estevan Moron
ALUNO: Lourival Aparecido de Góis

São Carlos
Agosto/2002

UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

“IMPLEMENTAÇÃO DOS SERVIÇOS DE COMUNICAÇÃO E MONITORAÇÃO EM UM AMBIENTE PARA DESENVOLVIMENTO DE SISTEMAS DISTRIBUIDOS”

Lourival Aparecido de Góis

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

SÃO CARLOS – SP

2002

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

G616is

Góis, Lourival Aparecido.
Implementação dos serviços de comunicação e monitoração em um ambiente para desenvolvimento de sistemas distribuídos / Lourival Aparecido Góis. -- São Carlos : UFSCar, 2003.
82 p.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2002.

1. Sistemas distribuídos. 2. Redes de comunicação de dados. 3. Monitoração de redes. 4. Arquitetura de redes de computadores. 5. AGAD. 6. TEV. 7. Programação em tempo real. I. Título.

CDD: 005.43 (20ª)

Dedico esse trabalho aos meus pais, José e Ordaliza, que sempre me incentivaram a buscar no estudo a base necessária para o meu futuro.

// AGRADECIMENTOS

Em primeiro lugar, à Deus;

Aos meus pais, cuja dedicação e zelo serviram de inspiração para o desenvolvimento deste trabalho;

A minha maior conquista, Eliana, que com seu amor, carinho e dedicação tornaram esse sonho uma realidade;

Ao meu filho Igor, minha maior criação;

A todos os meus amigos, pelo sorriso companheiro nos momentos de alegria e descontração, e pela palavra de apoio e conforto nas horas difíceis;

Ao Professor Dr. Célio Estevan Moron, grande incentivador, pela dedicação, seriedade e compreensão, meu sincero obrigado.

Lourival Aparecido de Góis

RESUMO

O ambiente TEV (*Teaching Environment for Virtuoso*) é um conjunto de ferramentas de ensino que suportam as fases de análise, projeto, implementação, depuração e testes de aplicações de tempo real que são executadas usando o *kernel* Virtuoso (*Virtual Single Processor Programming System*). Tal ambiente permite ao projetista da aplicação definir graficamente suas estruturas de dados através de um editor, escrever o código em linguagem C das suas tarefas e funções e, por fim, gerar os arquivos necessários para sua compilação. Estas facilidades resolvem grande parte dos problemas encontrados na geração de aplicações de tempo real, porém, as mesmas não dão suporte a aplicações distribuídas. O objetivo deste trabalho de mestrado é a modelagem e implementação de uma API (*Application Programming Interface*), baseada na pilha de protocolos TCP/IP, para ser utilizada em conjunto com o ambiente TEV de forma a prover funcionalidades relacionadas com a geração de aplicações de tempo real em uma estrutura de rede, bem como tratar problemas originados pela transparência existente na comunicação entre os protocolos e os processos destas aplicações. A partir desta API foi desenvolvido um gerador, denominado AGAD (Ambiente para Geração de Aplicações Distribuídas), para ser utilizado na implementação e monitoração das tarefas envolvidas com a comunicação. Este trabalho apresenta a modelagem da API, sua utilização na construção da ferramenta AGAD e a integração desta com o TEV.

ABSTRACT

The TEV (Teaching Environment for Virtuoso) environment is a set of teaching tools that support the phases of analysis, design, implementation, depuration and test of real-time applications which are executed using the Virtuoso (Virtual Single Processor Programming System) kernel. This environment allows the application designer to graphically define the application data structures through an editor, write the C language code of the tasks and functions of the application and, finally, generate the necessary files to compile the application. These facilities solve most of the problems while generating of real-time applications, however, they do not support distributed applications. This master research aims at modeling and implementing an API (Application Programming Interface), based on TCP/IP protocols, to be used together with the TEV environment in order to provide some functionality related to real-time application generation on a network structure as well as to deal with some problems related to transparency in the communication between the protocols and the application processes. Based on this API, a generator, called AGAD, was developed to be used in the implementation and monitoring of the tasks involved with the communication. This work presents the API modeling, its utilization on the creation of the AGAD and the integration of the with TEV.

LISTA DE ABREVIATURAS

API	Application Programming Interface
ARP	Address Resolution Protocol
ATEPC	Analisador do Tempo de Execução do Pior Caso
COM	Component Object Model
CORBA	Common Object Request Arquitecture Broker
DCOM	Distributed Component Object Model
DLL	Dynamic Linked Library
DNS	Domain Name System
EGP	Exterior Gateway Protocol
FTP	File Transfer Protocol
GGP	Gateway to Gateway Protocol
GPP	Gerador de Programas Paralelos
ICMP	Internet Control Message Protocol
IDL	Interface Definition Language
IIOP	Inter-ORB Protocol
IP	Internet Protocol
ISP	Internet Service Provider
ISR0	Interrupt Service Routine 0
ISR1	Interrupt Service Routine 1
LAN	Local Area Network
MAN	Metropolitan Area Network
OLE	Object Linking and Embedding
OMA	Object Management Arquitecture
OMG	Object Management Group
OSPF	Open Shortest Path First
PPP	Point-to-Point Protocol
RARP	Reverse Address Resolution Protocol
RAS	Remote Access Service
RIP	Routing Information Protocol
RPC	Remote Procedure Call
SDTR	Sistemas Distribuídos de Tempo Real
SNMP	Simple Network Management Protocol
SOTR	Sistemas Operacionais para Aplicações de Tempo Real
SP	Single Processor
STR	Sistemas de Tempo Real
TCP	Transmission Control Protocol
TEV	Teaching Environment for Virtuoso
ToS	Type of Service
UDP	User Datagram Protocol
VIRTUOSO	Virtual Single Processor Programming System
VSP	Virtual Single Processor
WAN	Wide Area Network

LISTA DE ILUSTRAÇÕES

FIGURA 01 – Camadas do Modelo de Referência OSI 8

FIGURA 02 – Arquitetura de Protocolos em Camadas 8

FIGURA 03 – Classificação dos Endereços por Sub-redes 15

FIGURA 04 - Componentes em Processos Diferentes 19

FIGURA 05 - Componentes em Processos e Máquinas Diferentes 20

FIGURA 06 - Arquitetura Cliente/Servidor em 2 níveis 27

FIGURA 07 - Arquitetura Cliente/Servidor Multinível 28

FIGURA 08 - Arquitetura Cliente/Servidor Par-Par 28

FIGURA 09 - Arquitetura Cliente/Servidor como Servidor de Arquivo 31

FIGURA 10 - Arquitetura Cliente / Servidor como Servidor de Banco de Dados 32

FIGURA 11 - Integração entre os processos clientes e servidores 32

FIGURA 12 – Primitivas de Comunicação Cliente/Servidor Via Sockets 51

FIGURA 13 – Modelagem Básica da API *Network* 52

FIGURA 14 – O AGAD e sua Conexão com o Sistema Operacional 61

FIGURA 15 – Ambiente para Geração de Aplicações Distribuídas 62

FIGURA 16 - Módulo Monitor 64

FIGURA 17 – Função do Monitor para Definição de Critérios para Captura de Pacotes 65

FIGURA 18 – Níveis de Programação do *Ker*nel Virtuoso 67

FIGURA 19 – Componentes do TEV 68

FIGURA 20 – Teaching Environment for Virtuoso 69

FIGURA 21 - Identificação dos Canais no Código Fonte do TEV 72

FIGURA 22 – Interação entre as Aplicações do TEV e do AGAD 73

FIGURA 23 – Mensagens Exibidas pela Ferramenta na Criação da Aplicação Servidor 76

FIGURA 24 – Seleção de canais de entrada e saída em aplicação desenvolvida no TEV 77

SUMÁRIO

Capítulo 1 1

1.1 Motivação *1*

1.2 Objetivos e Enfoques Utilizados para o Desenvolvimento do Projeto *3*
- 1.2.1 Desenvolvimento de uma API para a Implementação e Monitoração de Aplicações Cliente/Servidor 3
- 1.2.2 Desenvolvimento do AGAD (Ambiente para Geração de Aplicações Distribuídas) 3
- 1.2.3 Integração do AGAD com o TEV 3
- 1.2.3 Suporte a Execução de Aplicações Distribuídas 4
- 1.2.4 Uso de Técnicas de Modelagem 4

1.3 Resumo do Capítulo *5*

Capítulo 2 6

2.1 Redes de Computadores *6*

2.2 Protocolos e o Modelo de Referência ISO/OSI *7*

2.3 Descrevendo a Família de Protocolos TCP/IP *8*
- 2.3.1 Camada de Interface de Rede 9
- 2.3.2 Camada de Rede (IP) 9
  - 2.3.2.1 Internet Protocol (IP) 9
  - 2.3.2.2 Internet Control Message Protocol (ICMP) 10
  - 2.3.2.3 Address Resolution Protocol (ARP) 10
  - 2.3.2.4 Reverse Address Resolution Protocol (RARP) 11
  - 2.3.2.5 Roteamento 11
- 2.3.3 Camada de Transporte 12
  - 2.3.3.1 Transmission Control Protocol (TCP) 13
  - 2.3.3.2 User Datagram Protocol (UDP) 14
- 2.3.4 Camada de Aplicação 14
- 2.3.5 Endereçamento 15
  - 2.3.5.1 Mapeamento dos Endereços 15

2.4 Sistemas Distribuídos *16*
- 2.4.1 Tecnologias para a Implementação da Computação Distribuída 16
  - 2.4.1.1 RPC – Remote Procedure Call 17
  - 2.4.1.2 DCOM – Distributed Component Object Model 18
  - 2.4.1.3 CORBA - Common Object Request Broker Architecture 21

2.5 Arquitetura Cliente/Servidor *23*
- 2.5.1 - Modelos da Arquitetura Cliente/Servidor 26
  - 2.5.1.1 - Arquitetura Cliente/Servidor Simples 26
  - 2.5.1.2 - Arquitetura Cliente/Servidor em Dois Níveis 27
  - 2.5.1.3 - Arquitetura Cliente/Servidor Multinível 27
  - 2.5.1.4 - Arquitetura Cliente/Servidor Par-Par 28
- 2.5.2 – Processos em uma Arquitetura Cliente/Servidor 28
  - 2.5.2.1 - Processamento Distribuído 30
- 2.5.3 Camadas da Arquitetura Cliente / Servidor 30
- 2.5.4 IPC - Interprocess Communication 33
  - 2.5.4.1 Mecanismos de IPC 34
    - 2.5.4.1.1 Semáforos 34
    - 2.5.4.1.2. Troca de mensagens 35

2.5.4.1.3. Filas de mensagens 36
2.5.4.1.4 Sockets 37
2.5.4.2 Comunicação entre Processos de Aplicações Cliente / Servidor 38

2.6 Monitoração da Comunicação entre Aplicações Cliente/Servidor *39*
2.6.1 SNIFFERS - Ferramentas para Captura de Pacotes 40

2.7 Resumo do Capítulo *42*

Capítulo 3 43

3.1 Modelagem e Implementação da API Network *43*

3.2 Descrevendo a Interação entre Processos através dos Protocolos *44*

3.3 Programando Sockets *45*
3.3.1 Criação de um *Socket* 45
3.3.2 Fechamento de um Socket 46
3.3.3 Especificação de um Endereço Local 46
3.3.4 Especificação de um Comprimento de Fila para um Servidor 47
3.3.5 Preparação do Servidor para Aceitar Conexões 47
3.3.6 Vinculação de Sockets a Endereços Destino 47
3.3.7 Transmissão de Dados Através de um Socket 48
3.3.8 Recepção de Dados Através de um Socket 49
3.3.9 Obtenção de Endereços Locais e Remotos de Sockets 49
3.3.10 Outros Procedimentos Relacionados a Sockets 50
3.3.11 Modelagem da Conexão Via Socket entre Aplicações Cliente/Servidor 50

3.4 Modelagem da API Network *52*
3.4.1 Pacote GAD 52
3.4.1.1 Classe GetXbY 52
3.4.1.2 Classe Finish 53
3.4.1.3 Classe Function 53
3.4.1.4 Classe Open 54
3.4.1.5 Classe Transfer 55
3.4.1.6 Diagrama de Classes do Pacote GAD 56
3.4.2 Pacote MONITOR 56
3.4.2.1 Classe CDigDispositivo 56
3.4.2.2 Classe CDlgFiltro 57
3.4.2.3 Classe CCapturaPacote 57
3.4.2.4 Classe CEstatistica 58
3.4.2.5 Classes CPacoteIp, CPacoteTCP, CPacoteUDP e CPacoteICMP 58
3.4.2.6 Diagrama de Classes do Pacote MONITOR 59

3.5 Resumo do Capítulo *60*

Capítulo 4 61

4.1 AGAD - Ambiente para Geração de Aplicações Distribuídas *61*

4.2 Descrevendo o AGAD *62*
4.2.1 Monitorando o Tráfego entre Aplicações Distribuídas 63

4.3 Resumo do Capítulo *65*

Capítulo 5 66

5.1 Descrevendo a Interação do AGAD com o Ambiente Visual TEV *66*

5.2 TEV – Teaching Environment For Virtuoso *66*
5.2.1 Gerador de Programas Paralelos (GPP) 68

5.3 Mecanismos Utilizados no Compartilhamento de Dados entre Aplicações Distribuídos *70*
5.3.1 A Interação entre Processos Através dos Canais do Virtuoso 70
5.4 Implementação dos Mecanismos de Interação entre o AGAD e o TEV *71*
5.6 Resumo do Capítulo *74*

Capítulo 6 .... **75**

6.1 Gerando Aplicações AGAD Integradas ao TEV *75*
6.2 Implementando uma Aplicação Servidor *75*
6.2.1 Efetuando a conexão do Servidor Socket com uma Aplicação do TEV 76
6.3 Implementando uma Aplicação Cliente *77*
6.4 Resumo do Capítulo *78*

Capítulo 7 .... **79**

7.1 Conclusão *79*
7.2 Trabalhos Futuros *80*

REFERÊNCIAS BIBLIOGRÁFICAS **81**

Capítulo 1
INTRODUÇÃO

1.1 Motivação

No ambiente institucional tradicional, o processo de ensino-aprendizagem é projetado, gerenciado e implementado pelo professor, enquanto que em um ambiente institucional distribuído, este processo deve ser projetado pelo professor e gerenciado por um software, podendo ser compartilhado por professores, alunos e outras entidades, como editores e provedores de informação.

Nos últimos anos, os educadores testemunharam o rápido desenvolvimento das redes de computadores, o melhoramento no poder de processamento dos computadores pessoais e os avanços na tecnologia de armazenamento magnético. Estes avanços tornaram o computador uma ferramenta para o ensino, fornecendo meios novos e interativos para superar o tempo e a distância. As aplicações do computador no ensino encontram-se divididas em quatro categorias [1]:

1) Instrução Assistida por Computador (*Computer Attended Instruction - CAI*): consiste no uso do computador como uma ferramenta pedagógica de aprendizagem auto-suficiente para apresentar lições individuais e alcançar objetivos educacionais específicos, mas limitados. Há vários tipos de CAI, incluindo: exercícios práticos, tutoriais, simulações e jogos e solução de problemas;
2) Instrução Gerenciada por Computador (*Computer Managed Instruction - CMI*): fundamenta-se no uso da diversificação do computador, armazenagem e capacidade de interação com o usuário para organizar instruções e acompanhar o progresso dos estudantes. A instrução não precisa ser devolvida via computador, embora freqüentemente a CAI esteja combinada com a CMI;
3) Comunicação Mediada por Computador (*Computer Mediated Communication - CMC*): baseia-se no uso das aplicações do computador que facilitam a comunicação. Exemplos incluem o correio eletrônico, as videoconferências e as instruções através de avisos eletrônicos;

4) Multimídia Baseada em Computador (*Computer Based Multimedia - CBM*): compõe-se de ferramentas de computação poderosas, sofisticadas e flexíveis, aspectos que têm despertado a atenção de educadores nos últimos anos. O objetivo da multimídia baseada em computador é integrar várias ferramentas de voz, vídeo e tecnologias de computação em um sistema de acesso simples.

Os estudos do Grupo de pesquisadores de Sistemas Distribuídos do Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos (UFSCar), no uso do computador como auxiliar no ensino de sistemas distribuídos, resultaram no desenvolvimento de ferramentas assistidas e gerenciadas por computador, utilizadas para a geração do código fonte de sistemas paralelos de tempo real, executados com o suporte do *kernel* Virtuoso, propriedade da empresa *EONIC SYSTEMS*, com a qual o Departamento de Computação da UFSCar tem desenvolvido projetos conjuntos [13]. Essas ferramentas auxiliam no ensino de sistemas distribuídos através da depuração de erros e análise dos requisitos de tempo real. Fazem parte deste ambiente [10,11], um Gerador de Programas Paralelos, um Analisador do Tempo de Execução do Pior Caso, um Analisador de Escalonamento e um Depurador Paralelo.

O Gerador de Programas Paralelos, através de um modelo gráfico integrador de todas as ferramentas do ambiente visual, auxilia na geração do código fonte dos programas executados na máquina paralela através de grafos onde são representadas as estruturas de dados do programa paralelo e as operações de comunicação e sincronização relacionadas a essas estruturas. No entanto, esta ferramenta não oferece recursos quando ocorre a necessidade de se desenvolver programas que interajam entre si a partir de uma estrutura de rede, haja visto que as primitivas do *kernel* não executam chamadas diretamente para os protocolos. Levando-se em consideração que a ferramenta é executada em uma estrutura composta pelo Sistema Operacional Windows NT [14,19,22] e pela pilha de protocolos TCP/IP [2,4], foi definido que seriam implementados alguns componentes para serem utilizados em conjunto com a ferramenta TEV, que comportassem todas as primitivas necessárias à comunicação entre as tarefas, utilizando para isto as primitivas básicas do *kernel* Virtuoso [13], conforme descritas no decorrer do levantamento bibliográfico. Estes componentes deverão

abranger, além do que foi definido acima, alguns critérios que possibilitem a monitoração e verificação das etapas pertinentes à comunicação. Sendo assim, foram estudados vários conceitos relacionados com este projeto, tais como protocolos TCP/IP, sistemas distribuídos, *sockets*, sistemas operacionais e aplicações de tempo real.

1.2 Objetivos e Enfoques Utilizados para o Desenvolvimento do Projeto

1.2.1 Desenvolvimento de uma API para a Implementação e Monitoração de Aplicações Cliente/Servidor

A API *Network* é manipulada pelo desenvolvedor de aplicações baseadas no paradigma cliente/servidor como um conjunto de classes formadas basicamente por estruturas de dados que tratam comunicação, execução de código, processamento distribuído e monitoração de tráfego. Abaixo dessa camada visível ao desenvolvedor, existem classes que gerenciam o funcionamento da API *Network*, com detalhes de implementação transparentes aos usuários.

1.2.2 Desenvolvimento do AGAD (Ambiente para Geração de Aplicações Distribuídas)

A partir da API Network, foi modelado o AGAD, um utilitário responsável pela geração de aplicações distribuídas baseadas no paradigma cliente/servidor, que utilizam o mecanismo de comunicação entre seus processos denominado sockets. Integrado ao AGAD encontra-se um monitor de tráfego para ser utilizado na verificação das informações trafegadas através da captura de pacotes na rede.

1.2.3 Integração do AGAD com o TEV

Visando a integração das aplicações geradas a partir do AGAD com as aplicações geradas a partir do TEV, foi estudado um modelo que

permitisse essa comunicação. Esse estudo foi feito buscando garantir as funcionalides individuais de cada ambiente, permitindo que com isso eles possam ser utilizados em conjunto ou em separado.

1.2.3 Suporte a Execução de Aplicações Distribuídas

Esse trabalho visa facilitar a criação de aplicações que possam ser distribuídas em diversos computadores conectados a uma rede. Através da API *Network*, o desenvolvedor de programas cliente/servidor poderá especificar as funções a serem utilizadas e definir em quais computadores estas aplicações serão executadas.

O suporte para a distribuição de tarefas em processadores distintos é uma extensão adicionada ao *kernel* de tempo real Virtuoso, que trabalha somente em placas paralelas de processamento dedicado e não em redes de computadores. Esta característica de distribuição foi implementada usando *sockets* [3]. A opção por este mecanismo de comunicação deve-se à diminuição da latência do envio e recebimento de mensagens na rede proporcionadas por esta abordagem, o que é particularmente importante em sistemas de tempo real executados sob este *kernel* [3,8,13].

1.2.4 Uso de Técnicas de Modelagem

Para a documentação deste projeto, utilizou-se a linguagem de modelagem UML (*Unifield Modeling Language*) [16], que une as melhores características de vários métodos de modelagem, gerada através da ferramenta *Rational Rose* [16]. Com a aplicação desta técnica de modelagem, buscou-se eliminar erros no projeto da API *Network*, eliminar as ambigüidades e, por fim, documentá-la. Com isto, através de uma modelagem clara e concisa, procurou-se modelar uma API na forma de um objeto de fácil manipulação para estudos futuros, possibilitando o aprendizado e a reusabilidade das técnicas desenvolvidas neste projeto.

1.3 Resumo do Capítulo

Este capítulo demonstrou o interesse do grupo de Sistemas Distribuídos e Paralelos no desenvolvimento de um projeto para a implementação de uma ferramenta para auxiliar na geração de aplicações baseadas no paradigma cliente/servidor e com conexão via *socket* por ser este o mecanismo mais adequado ao projeto por sua simplicidade e facilidade de implementação. Apresentou-se também, os objetivos e enfoques utilizados para o desenvolvimento deste trabalho de mestrado que visa a construção de uma API (*Application Programming Interface*) e sua utilização na ferramenta denominada AGAD (Ambiente para Geração de Aplicações Distribuídas), bem como, sua integração com o TEV (*Teaching Environment for Virtuoso*), um ambiente construído para o desenvolvimento de aplicações de tempo real sob o *kernel* Virtuoso.

No próximo capítulo serão descritos vários conceitos básicos e fundamentais que foram necessários para o desenvolvimento deste trabalho.

Capítulo 2
CONCEITOS FUNDAMENTAIS

2.1 Redes de Computadores

Uma Rede de Computadores é formada por um conjunto de módulos processadores capazes de trocar informações e compartilhar recursos, interligados por um sistema de comunicação. Este sistema é constituído de um arranjo topológico interligando os vários módulos processadores através de enlaces físicos e de um conjunto de regras com o fim de organizar a comunicação. Redes de computadores são ditas confinadas quando as distâncias entre os módulos processadores são menores que alguns poucos metros. Redes Locais de Computadores são sistemas cujas distâncias entre os módulos processadores se enquadram na faixa de alguns poucos metros a alguns quilômetros. Sistemas cuja dispersão é maior do que alguns quilômetros são chamadas Redes Geograficamente Distribuídas [6,12].

As redes locais surgiram para viabilizar a troca e o compartilhamento de informações e dispositivos periféricos, preservando a independência das várias estações de processamento e permitindo a integração em ambientes de trabalho cooperativo. Pode-se caracterizar uma rede local com sendo uma rede que permite a interconexão de equipamentos de comunicação de dados numa pequena região, cujas distâncias oscilem entre 100m e 25Km, embora as limitações associadas às técnicas utilizadas em redes locais não imponham limites a essas distâncias. Outras características típicas encontradas e comumente associadas a redes locais são: altas taxas de transmissão e baixas taxas de erro; outra característica é que em geral, elas são de propriedade privada.

Quando a distância de ligação entre vários módulos processadores começa a atingir distâncias metropolitanas, chamamos esses sistemas não mais de rede locais, mas de Redes Metropolitanas (*Metropolitan Area Networks - MANs*).

Uma rede metropolitana apresenta características semelhantes às redes locais, sendo que as MANs em geral, cobrem distâncias maiores que as LANs, operando em velocidades maiores.

As Redes Geograficamente Distribuídas surgiram da necessidade de se compartilhar recursos especializados por uma comunidade maior de usuários geograficamente dispersos. Por terem um custo de comunicação bastante elevado, tais redes são em geral públicas, isto é, o sistema de comunicação, chamado sub-rede de comunicação, é mantido gerenciado e de propriedade pública. Face a várias considerações em relação ao custo, a interligação entre os diversos módulos processadores em uma tal rede determinará a utilização de um arranjo topológico específico e diferente daqueles utilizados em redes locais. Ainda por problemas de custo, as velocidades de transmissão empregadas são baixas: da ordem de algumas dezenas de kilobits/segundo (embora alguns enlaces cheguem hoje à velocidade de megabits/segundo). Por questão de confiabilidade, caminhos alternativos devem ser oferecidos de forma a interligar os diversos módulos.

2.2 Protocolos e o Modelo de Referência ISO/OSI

Um protocolo pode ser definido como um conjunto de regras e procedimentos que regulamentam o processo de comunicação entre computadores e aplicações em uma rede. Visando a padronização dessas regras e procedimentos, foi constituída pela ISO (*International Standards Organization*), em março de 1977, um grupo de trabalho para estudar mecanismos que pudessem ser adotados como padrões na interconexão de sistemas de computação. Inicialmente, foi definida uma arquitetura geral denominada Modelo de Referência OSI (*Open System Interconection*), para servir de base nas conexões entre sistemas abertos [6].

O modelo OSI possui sete camadas, conforme apresentado na FIGURA 01. Cada camada especifica um grupo de tarefas da comunicação. A norma descreve o escopo estrutural de cada camada e os requisitos para a interface com as camadas adjacentes.

Camada de Aplicação	Seleção de serviços apropriados à aplicação
Camada de Apresentação	Formatação/Reformatação de dados. Ex.: Criptografia
Camada de Sessão	Interface do usuário para o estabelecimento de conexões
Camada de Transporte	Controle de intercâmbio de mensagens entre usuários
Camada de Rede	Controle de intercâmbio de mensagens na rede
Camada de Enlace	Transmissão livre de erros
Camada de Física	Interface elétrica para transmissão bit a bit

FIGURA 01 – Camadas do Modelo de Referência OSI

A FIGURA 02 mostra que quando a mensagem passa da camada n +1 para a camada n são acrescidos outros dados relevantes à camada n . Estes dados são retirados quando a mensagem chega na camada de mesmo nível na estação destino. Estes acréscimos podem ser informações tais como: tipo da mensagem, endereços, tamanho da mensagem, código de detecção de erro, entre outros.

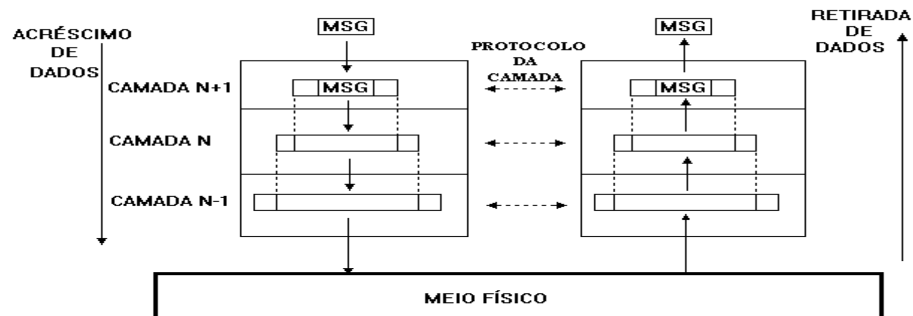


FIGURA 02 – Arquitetura de Protocolos em Camadas

O modelo ISO/OSI especifica todas as primitivas de comunicação para que haja troca de mensagens entre as camadas [2,6]. Cada camada adiciona um cabeçalho para que haja uma identificação na máquina destino.

2.3 Descrevendo a Família de Protocolos TCP/IP

O modelo TCP/IP [2,4] não pertence ao modelo ISO/OSI, sendo constituído basicamente por duas camadas: a camada de rede e a camada de transporte. Tanto a camada de aplicação quanto a camada de interface de rede não possuem uma norma definida, devendo a camada de aplicação utilizar serviços da

camada de transporte a ser definida adiante, e a camada de interface de rede prover a interface dos diversos tipos de rede com o protocolo (promovendo em conseqüência a interoperação com as diversas arquiteturas de rede, *Ethernet*, *Token Ring*, ATM, entre outras).

2.3.1 Camada de Interface de Rede

Também chamada camada de abstração de hardware, tem como função principal a interface do modelo TCP/IP com os diversos tipos de redes (X.25, ATM, FDDI, *Ethernet*, *Token Ring*, *Frame Relay*, sistema de conexão ponto-a-ponto SLIP, etc.). Como há uma grande variedade de tecnologias de rede que utilizam diferentes velocidades, protocolos e meios transmissão, esta camada não é normatizada pelo modelo, o que resulta numa das grandes virtudes do modelo TCP/IP: a possibilidade de interconexão e interoperação de redes heterogêneas.

2.3.2 Camada de Rede (IP)

A camada de rede é a primeira especificada no modelo. Também conhecida como camada Internet, é responsável pelo endereçamento, roteamento dos pacotes, controle de envio e recepção (erros, bufferização, fragmentação, seqüência, reconhecimento), entre outros.

Dentre os protocolos da Camada de Rede, destaca-se inicialmente o IP (*Internet Protocol*), além do ARP, ICMP, RARP e dos protocolos de roteamento (RIP ,IGP, OSPF, Hello, EGP e GGP). A camada de rede é uma camada não orientada á conexão, portanto comunica-se através de datagramas. Mais informações a respeito dos protocolos desta camada poderão ser obtidas adiante.

2.3.2.1 Internet Protocol (IP)

A função básica do protocolo IP é o transporte dos blocos de dados por entre as sub-redes até chegar ao destinatário. Durante o tráfego pelas sub-redes, existem componentes denominados *gateways,* que encaminham o

datagrama IP para outras sub-redes ou para o destinatário, se este fizer parte da sub-rede a que o *gateway* está conectado.

Por limitação tecnológica, algumas sub-redes têm capacidade apenas para trafegar pacotes menores (volume de dados menor). Assim, o roteador fragmenta o datagrama original em datagramas menores, que serão restabelecidos futuramente, quando possível.

2.3.2.2 Internet Control Message Protocol (ICMP)

O protocolo ICMP é utilizado para transmissão de mensagens de controle ou de ocorrência de problemas. Utiliza o protocolo IP para o transporte das mensagens. Geralmente as mensagens ICMP são geradas pelos *gateways*, podendo também ser geradas pela estação destinatária.

No caso de problemas com datagramas enviados pela estação de origem, o ICMP inclui no seu datagrama de ocorrências o cabeçalho, além de 64 bits iniciais dos dados do datagrama IP que originou o erro.

As ocorrências do ICMP podem ser:

1) Destinatário inacessível;
2) Ajuste de fonte — Solicita à estação a redução da taxa de emissão de datagramas;
3) Redireção — Rota mais adequada para a estação destinatária (para atualização da tabela de endereço dos roteadores);
4) Eco e Resposta de Eco;
5) Tempo excedido;
6) Problemas de parâmetros;
7) Marca de Tempo e Resposta à Marca de Tempo;
8) Solicitação de informações e Respostas de Informações;
9) Solicitação de Máscara de endereço e Resposta à Máscara de Endereço.

2.3.2.3 Address Resolution Protocol (ARP)

São utilizados para o mapeamento dinâmico do endereço IP. Quando inicializadas, as estações não possuem uma tabela de endereços IP/físico

armazenada. Em vez disso, para cada endereço IP que não esteja na tabela do roteador, o protocolo ARP manda uma solicitação via *broadcast* do endereço físico para o endereço IP determinado. O destinatário que tiver o endereço IP informado responde à máquina solicitante, seu endereço físico. Nessa ocasião, tanto a tabela da máquina origem quanto a da máquina destinatária são atualizadas com os endereços.

2.3.2.4 Reverse Address Resolution Protocol (RARP)

De forma inversa ao ARP, o RARP procura um endereço IP relacionado a um endereço físico determinado. Geralmente quem mais utiliza tal protocolo são as estações de rede *diskless* que possuem apenas o endereço físico, durante o processo de inicialização.

Para que o RARP funcione, é necessário ao menos um servidor RARP, que possui informações de mapeamento de todas as estações da rede.

Da mesma forma que o ARP, o RARP envia uma mensagem broadcast solicitando o endereço IP. Será pesquisado nas tabelas dos servidores o endereço solicitado, sendo então devolvida uma mensagem RARP contendo a informação solicitada.

Caso haja mais de um servidor RARP, um deles é determinado como prioritário, onde será feita a primeira pesquisa. Se dentro de um intervalo de tempo não houver respostas, outros servidores iniciarão a pesquisa.

2.3.2.5 Roteamento

É o processo de escolha do caminho pelo qual o pacote deve chegar à estação destinatária. O roteamento pode ser direto ou indireto, conforme detalhado abaixo:

1) Roteamento Direto: O roteamento direto ocorre quando a estação destinatária do datagrama está na mesma sub-rede física da estação origem. A checagem é feita comparando-se o endereço IP do emissor e do destinatário constantes no

datagrama IP. Nesse caso, o conteúdo do datagrama recebe o endereço físico da estação e é enviado diretamente pela mesma sub-rede;

2) Roteamento Indireto: No caso do roteamento indireto, o emissor deve enviar para o *gateway* o datagrama com o endereço IP do destinatário. O *gateway* verificará se o destinatário pertence a uma das sub-redes a ele conectadas e, em caso positivo, enviará o pacote diretamente para a estação. Caso o *gateway* não localize o destinatário como um membro de uma das sub-redes a ele conectadas, ele enviará o pacote para outro *gateway* (de acordo com sua tabela de roteamento), que verificará o mesmo, e assim por diante até encontrar o destinatário ou terminar o tempo de vida do pacote.

2.3.3 Camada de Transporte

A camada de transporte é uma camada fim-a-fim, isto é, uma entidade desta camada no equipamento de origem, só se comunica com a sua entidade-par do equipamento destinatário. É nesta camada que se faz o controle da conversação entre as aplicações intercomunicadas da rede.

A camada de transporte utiliza dois protocolos: o TCP e o UDP. O primeiro é orientado à conexão e o segundo é não-orientado à conexão. Ambos os protocolos podem servir a mais de uma aplicação simultaneamente.

O acesso das aplicações à camada de transporte é feito através de *portas* que recebem um número inteiro para cada tipo de aplicação, podendo também tais portas serem criadas ao passo em que novas necessidades vão surgindo com o desenvolvimento de novas aplicações.

A maneira como a camada de transporte transmite dados das várias aplicações simultâneas é por intermédio da multiplexação, onde várias *mensagens* são repassadas para a camada de rede (especificamente ao protocolo IP) que se encarregará de empacotá-las e mandar para uma ou mais interfaces de rede. Chegando ao destinatário, o protocolo IP repassa para a camada de transporte que demultiplexa para as portas (aplicações) específicas. A seguir serão detalhados os principais protocolos envolvidos nesta camada.

2.3.3.1 Transmission Control Protocol (TCP)

É o protocolo TCP que faz a comunicação fim-a-fim da rede. É orientado à conexão e altamente confiável, independente da qualidade de serviços das sub-redes que servem de caminho. Para a confiabilidade de transmissão, garante a entrega das informações na seqüência em que lhe foi fornecida, sem perda nem duplicação. Principais funções :

1) Transferência de dados — Através de mensagens de tamanho variável em *full-duplex*;
2) Transferência de dados urgentes — Informações de controle, por exemplo;
3) Estabelecimento e liberação de conexão — Antes e depois das transferências de dados, através de um mecanismo chamado *three-way-handshake*;
4) Multiplexação: As mensagens de cada aplicação simultânea são multiplexadas para repasse ao IP. Ao chegar ao destino, o TCP demultiplexa as mensagens para as aplicações destinatárias;
5) Segmentação: Quando o tamanho do pacote IP não suporta o tamanho do dado a ser transmitido, o TCP segmenta (mantendo a ordem) para posterior remontagem na máquina destinatária;
6) Controle do fluxo: Através de um sistema de buferização denominada *janela deslizante*, o TCP envia uma série de pacotes sem aguardar o reconhecimento de cada um deles. Na medida em que recebe o reconhecimento de cada bloco enviado, atualiza o *buffer* (caso o reconhecimento seja positivo) ou reenvia (caso o reconhecimento seja negativo ou não reconhecimento após um *timeout*);
7) Controle de erros: Além da numeração dos segmentos transmitidos, vai junto com o *header* uma soma verificadora dos dados transmitidos (*checksum*), assim o destinatário verifica a soma com o cálculo dos dados recebidos;
8) Precedência e segurança: Os níveis de segurança e precedência são utilizados para tratamento de dados durante a transmissão.

2.3.3.2 User Datagram Protocol (UDP)

O UDP é um protocolo mais rápido do que o TCP, pelo fato de não verificar o reconhecimento das mensagens enviadas. Por este mesmo motivo, não é confiável como o TCP.

O protocolo é não-orientado à conexão além de não prover muitas funções: não controla o fluxo podendo os datagramas serem transmitidos fora de seqüência ou, até mesmo, não chegarem ao destinatário. Opcionalmente pode conter um campo *checksum*, sendo que os datagramas que apresentarem divergência nesse campo no destino, serão descartados, cabendo à aplicação recuperá-los.

2.3.4 Camada de Aplicação

É formada pelos protocolos utilizados pelas diversas aplicações do modelo TCP/IP. Esta camada não possui um padrão comum. O padrão estabelece-se para cada aplicação, isto é, o FTP e o TELNET possuem seu próprio protocolo, bem como o SNMP, GOPHER, DNS, etc.

É na camada de aplicação que se estabelece o tratamento das diferenças entre representação de formato de dados. O endereçamento da aplicação na rede é provido através da utilização de portas para comunicação com a camada de transporte. Para cada aplicação existe uma porta predeterminada.

As aplicações, no modelo TCP/IP, não possuem uma padronização comum. Cada uma possui um RFC (*Request for Comment*) próprio. O endereçamento das aplicações é feito através de portas (chamadas padronizadas aos serviços dos protocolos TCP e UDP), por onde são passados as mensagens. Como já mencionado, é na camada de Aplicação que se trata a compatibilidade entre os diversos formatos representados pelos variados tipos de estações da rede.

A comunicação entre as máquinas da rede é possibilitada através de primitivas de acesso das camadas UDP e TCP. Estas primitivas estão detalhadas mais adiante.

2.3.5 Endereçamento

O endereçamento de datagramas no modelo TCP/IP é implementado pela camada de rede (IP). Uma das informações de controle do datagrama é o endereço IP do destinatário e do emitente.

O endereço IP é formado por um número de 32 bits no formato *nnn.nnn.nnn.nnn* onde cada *nnn* pode variar de 0 até 255 (1 octeto = 8 bits). Os endereços possuem uma classificação que varia de acordo com o número de sub-redes e de *hosts*. Tal classificação tem por finalidade otimizar o roteamento de mensagens na rede, conforme demonstrado na FIGURA 03.

Classificação	1o. Octeto	2o. Octeto	3o. Octeto	4o. Octeto
Classe A	0 Rede	Máquina		
Classe B	1 0 Rede		Máquina	
Classe C	1 1 0 Rede			Máquina
Classe D	1 1 1 0 Broadcasting e Multicasting			
Classe E	1 1 1 1 0 Reservado para utilização futura			

FIGURA 03 – Classificação dos Endereços por Sub-redes

Os endereços são fornecidos por uma entidade central: NIC (*Network Information Center*), e devem ser únicos para cada estação (*host*).

Para o usuário dos serviços de rede, há uma forma mais simples de endereçamento, que é através do DNS (*Domain Name System*), o protocolo responsável pela associação de um nome de computador a um endereço IP.

2.3.5.1 Mapeamento dos Endereços

A única forma de comunicação entre duas máquinas é através do seu endereço físico. Como já visto, o modelo TCP/IP pode interligar redes heterogêneas, portanto não há uma padronização de endereços físicos que pudesse relacionar direta e unicamente uma máquina e seu endereço IP.

Para resolver estes problemas de mapeamento do endereço IP para o endereço físico da rede, é utilizado o protocolo ARP (*Address Resolution Protocol*), que encontra o endereço físico relacionado a um endereço IP

fornecido. De igual forma, em se querendo um endereço IP a partir do endereço físico, utiliza-se o protocolo RARP (*Reverse Address Resolution Protocol*).

Há duas categorias de mapeamento. Quando o tamanho do endereço físico é menor ou igual ao tamanho do endereço IP (32 bits), pode-se fazer o endereço físico e o endereço IP iguais. De outra forma, quando o tamanho do endereço físico é maior que o do endereço IP (o endereço *Ethernet*, por exemplo, tem 48 bits) é criada uma tabela de mapeamento.

2.4 Sistemas Distribuídos

O sistema distribuído é caracterizado pela existência de um relacionamento mais forte entre os seus componentes, em que geralmente os sistemas operacionais são os mesmos. Podem ser considerados também como a evolução para os sistemas fortemente acoplados, onde uma aplicação pode ser executada por qualquer processador. Os sistemas distribuídos permitem que uma aplicação seja dividida em diferentes partes que se comunicam através de linhas de comunicação, e cada parte pode ser processada em um sistema independente. O objetivo do sistema distribuído é criar a ilusão de que toda rede de computadores nada mais é do que um único sistema de tempo compartilhado, em vez de um conjunto de máquinas distintas [7,17].

2.4.1 Tecnologias para a Implementação da Computação Distribuída

Quando se deseja implementar e fazer uso de aplicações distribuídas, existem várias alternativas que podem ser utilizadas, ficando a escolha sob a responsabilidade do projetista, que deverá selecioná-la a partir da natureza da aplicação e os recursos envolvidos, como sistema operacional, linguagem de programação, entre outros. A seguir, estão relacionadas algumas dessas alternativas.

2.4.1.1 RPC – Remote Procedure Call

O RPC é um modelo de chamada de procedimento remoto, que por sua vez é similar ao modelo de chamada de procedimento local. No caso local, o objeto que realiza a chamada coloca os argumentos para o procedimento em uma localização específica. Então, o controle é transferido ao procedimento e depois eventualmente ele ganha o controle novamente. Nesse ponto, os resultados do procedimento são extraídos da localização específica e o objeto continua sua execução.

Um *thread* de controle passa por dois processos. O processo que executa a chamada primeiro envia uma mensagem ao processo servidor e então espera por um mensagem de resposta. A mensagem de chamada inclui os parâmetros do procedimento e a mensagem de resposta inclui os resultados da chamada. Quando a resposta é recebida, os resultados são extraídos da mensagem e o processo que executou a chamada continua sua execução.

Do lado do servidor, existe um processo adormecido esperando por uma mensagem de chamada. Quando ela chega, o processo servidor extrai os parâmetros do procedimento, calcula os resultados, envia uma mensagem de resposta e volta a ficar adormecido esperando por outra mensagem de chamada.

Vale lembrar que a descrição acima é apenas uma das possibilidades de implementação do modelo RPC, tendo em vista que o modelo RPC da Sun Microsystems não faz restrições a outros modelos que apresentam a mesma funcionalidade. Por exemplo, em uma implementação, as chamadas RPC podem ser assíncronas, ou seja, o cliente pode executar outras tarefas enquanto espera pela resposta do servidor. Outra possibilidade seria o servidor criar um processo separado só para processar as chamadas dos clientes, assim o servidor original ficaria livre para receber outras chamadas.

Existem algumas diferenças importantes entre os procedimentos locais e remotos. São elas:

1) Tratamento de erros - falhas ocorridas no servidor remoto ou na rede devem ser tratadas;

2) Variáveis Globais e Efeitos Colaterais - tendo em vista que o servidor não tem acesso à faixa de endereços do cliente, argumentos escondidos não podem ser passados como variáveis globais ou retornados como efeitos colaterais;
3) Performance - procedimentos remotos em geral operam em velocidades algumas ordens de magnitude menores que chamadas a procedimentos locais;
4) Autenticação - a autenticação se faz necessária tendo em vista que chamadas a procedimentos remotos trafegam em redes inseguras.

A conclusão é que apesar de existirem ferramentas para automatizar a criação de bibliotecas de clientes e servidores para um determinado serviço, os protocolos devem ser projetados com muito cuidado e sempre observando as considerações acima.

2.4.1.2 DCOM – Distributed Component Object Model

O COM *(Component Objetct Model)* é o padrão pelo qual componentes de software podem utilizar ou serem utilizados uns pelos outros, integrando recursos entre diversos aplicativos.

É uma tecnologia que foi desenvolvida pela Microsoft Corporation no ano de 1993. Sua utilização está voltada para dar suporte orientado a objetos a diferentes partes de softwares que precisem se comunicar. Essa comunicação pode ser feita por objetos em execução no mesmo processo ou não, porém no mesmo processador.

Objetos COM podem ser desenvolvidos em linguagens distintas, sendo que, entre elas, as principais são: Visual Basic, Delphi, Java e C ++. No entanto, mesmo quando criados em linguagens diferentes, eles podem se comunicar. Na verdade um objeto COM (cliente COM) não toma conhecimento de que linguagem o objeto, com o qual ele queira se comunicar (componente COM), foi desenvolvido, assim como não sabe se estão utilizando a mesma *DLL* ou estão em processos diferentes.

A forma de comunicação mais simples é quando o objeto cliente e o componente estão no mesmo processo e, portanto, não há necessidade de qualquer recurso entre eles.

Na FIGURA 04, apresenta-se o funcionamento da comunicação entre dois objetos em processos diferentes. O COM intercepta a chamada do cliente ao servidor, padroniza-a e direciona-a até o componente em um outro processo. Esta padronização de direcionamento da chamada ocorre nas camadas que compõem o modelo COM. Primeiramente pela camada de segurança que valida o processo de comunicação, em seguida pela camada DCE RPC (*Distributed Computing Environment*), que irá concretizar a comunicação com o servidor através do LPC (*Local Procedure Call*). Esta mesma operação acontece de maneira inversa até chegar ao componente propriamente dito.

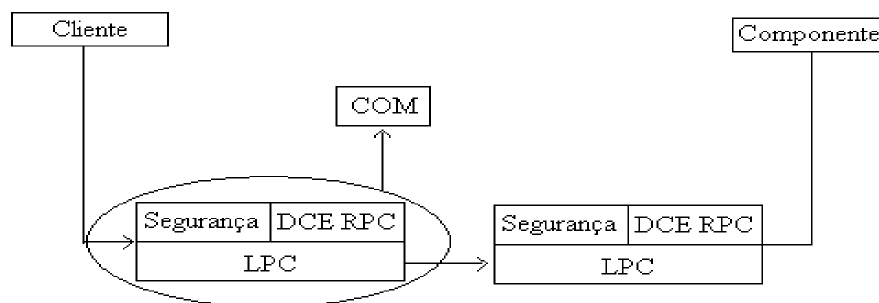


FIGURA 04 - Componentes em Processos Diferentes

O DCOM tem suas raízes nas tecnologias de objeto da Microsoft: o COM*,* ***OLE*** *(Object Linking and Embedding)* e *ActiveX* (COM habilitado para a Internet).

O DCOM é a versão distribuída da tecnologia COM. Ele possibilita a comunicação de objetos em processos diferentes e em execução em computadores distintos, interconectados numa rede, ou seja, o DCOM é uma evolução do COM, dispõe de suas mesmas características somadas à possibilidade de comunicação em máquinas distintas.

Deve-se considerar COM e DCOM como uma única tecnologia, que fornece uma escala de serviços para a interação de objetos. Se no momento de uma comunicação for necessário utilizar o padrão COM, ele será utilizado, porém se for necessária a tecnologia DCOM, esta também será utilizada. Ou seja, as duas tecnologias apresentam-se fundidas em um único *runtime***,** sendo que este fornece acesso local e remoto.

Falou-se anteriormente sobre o funcionamento da comunicação entre objetos num mesmo processador e processos diferentes, agora será explicada a comunicação entre objetos em máquinas e processos diferentes.

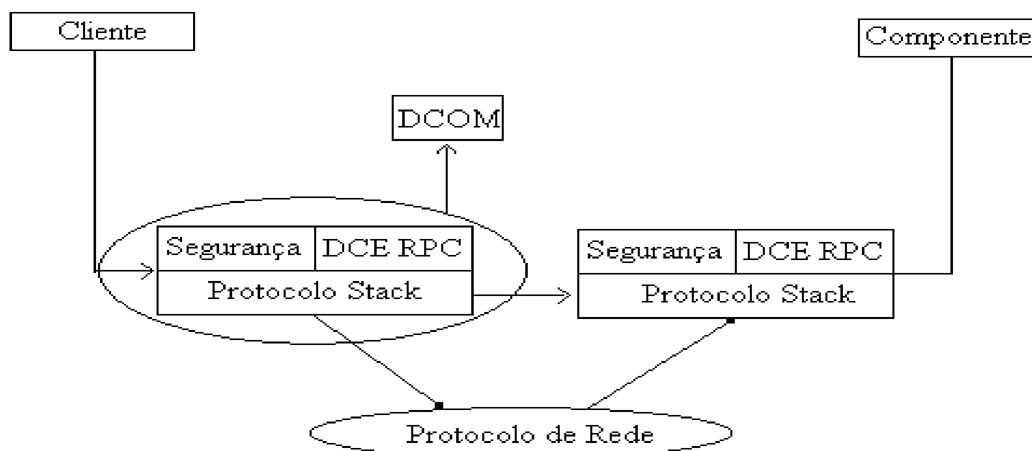


FIGURA 05 - Componentes em Processos e Máquinas Diferentes

Na FIGURA 05 um objeto cliente faz uma requisição. Esta é interceptada pelo COM, passa pela camada de segurança e chega até o DCE RPC. Neste momento já se sabe que o servidor encontra-se numa outra estação da rede; o DCE RPC comunica-se então com o servidor através do protocolo *Stack*. No servidor a mesma operação acontecerá inversamente até que a mensagem chegue ao objeto requisitado.

No momento em que uma comunicação como essa acontece, classificam-se os objetos como distribuídos COM ou DCOM. O funcionamento é quase o mesmo que acontece se estivessem numa mesma máquina. Na verdade nem os objetos, nem o programa sabem como essa comunicação é realizada, ela é totalmente transparente. Os objetos podem estar numa mesma máquina ou distantes, o tratamento para eles será o mesmo; a tecnologia DCOM fará a comunicação sem que se perceba que a distância é bem maior do que parece. Se no momento de chamar e padronizar a requisição ele perceber que o servidor está em uma máquina distinta, não importando o local, ele acionará um protocolo que irá auxiliar o DCE RPC na comunicação pela rede. Caso contrário a comunicação acontecerá apenas pelo DCE RCP e o LPC.

2.4.1.3 CORBA - Common Object Request Broker Architecture

O padrão CORBA [15] é um modelo proposto pela OMG, com o objetivo de promover na teoria e na prática, a tecnologia dos objetos de forma distribuída, ou seja, é uma estrutura comum para o desenvolvimento independente de aplicações, usando técnicas de orientação a objeto em redes de computadores heterogêneas. Visa diminuir consideravelmente os custos, reduzir a complexidade e proporcionar caminhos para o surgimento de novas aplicações a partir dos conceitos propostos pela OMG. O CORBA, resumidamente, propõe a interoperabilidade local ou remota entre aplicações, independente das linguagens de programação em que foram desenvolvidas e sobre quais plataformas serão executadas.

A OMG (*Object Management Group*) é uma organização formada por empresas dos diferentes ramos da informática (venda, desenvolvimento, produção). Inicialmente somavam-se treze empresas, hoje já são setecentas e cinqüenta, entre elas estão: DEC, Cannon, IBM, Sun, Apple e outras. Essas empresas trabalham sem fins lucrativos, para promover a criação e elaboração de modelos e padrões que proporcionem a interoperabilidade entre aplicações que usam tecnologia orientada a objeto.

A primeira versão do CORBA, a 1.1, surgiu em 1991, momento este, em que se definiu a IDL (*Interface Definition Language*) e a API (*Application Programming Interface*). Porém, a interoperabilidade entre os objetos desenvolvidos em linguagens de diferentes fabricantes ocorreu somente em 1994, com a segunda versão do CORBA, a 2.0, quando implementou-se no mesmo o IIOP (*Inter-ORB Protocol*).

A aplicação do modelo CORBA acontece para promover a intercomunicação de objetos distribuídos, a fim de executar alguma tarefa. Entretanto, o CORBA é apenas uma parte de uma outra tecnologia também proposta pela OMG, a *OMA (Object Management Architecture*), que compreende seus componentes, descritos da seguinte maneira:

1) Orientados ao Sistema: ORB's, Objetos de Serviços;
2) Orientados à Aplicação: Objetos de Aplicação, Facilidades Comuns;

3) *ORB (Object Request Broker):* componente definidor do barramento comum para a troca de mensagens entre os objetos;
4) Objetos de Serviços *(Common Object Services):* serviços implementados por objetos do sistema, utilizados para ampliar a funcionalidade do barramento de objetos (ORB);
5) Facilidades Comuns (*Common Facilites*): definem facilidades e interfaces no nível de aplicação;
6) Objetos de Aplicação (*Application Objects*): são os objetos propriamente ditos.

O ORB é a base do OMA, por conseqüência, também do CORBA, e exerce o papel de *middleware* entre os componentes. O ORB é responsável por toda comunicação e interação entre os objetos; ele intercepta a chamada e fica responsável por encontrar um objeto que atenda às necessidades do pedido. Encontrando o objeto, o ORB passa os parâmetros para o mesmo, invoca os métodos necessários dele e retorna para o objeto que solicitou os resultados de todo esse procedimento. Dessa maneira, o usuário não precisa se preocupar onde tal objeto está localizado, em que sistema operacional ele roda ou qual programa foi usado para desenvolvê-lo.

Os Objetos de Serviços gerenciam os objetos (criação, controle, rastreamento). Quanto aos Objetos de Aplicação e Facilidades Comuns, pode-se dizer que são componentes que estão mais diretamente relacionados com o usuário final, estando intimamente ligados com as invocações dos serviços que o sistema de cada usuário necessite.

Objetos distribuídos CORBA, especificamente, são pequenas partes inteligentes de códigos de um sistema maior, onde essas pequenas partes estão presentes em algum lugar da rede e apresentam-se como componentes binários que podem ser acessados por objetos clientes remotos ou não, por meio de métodos de invocação.

Para criar um objeto, utiliza-se uma linguagem de alto nível para escrever o código do objeto, como C, C++, Java, Smaltalk e Ada. A seguir o mesmo é compilado utilizando uma linguagem definida na especificação do CORBA (IDL).

Para a ocorrência das características conhecidas dos objetos distribuídos, o CORBA dispõe de um meio que padroniza os objetos, permitindo sua comunicação: a utilização de interfaces. Cada objeto tem uma interface definida, que deve conhecer e requisitar um serviço a outro objeto. Dessa maneira, não se faz necessário saber detalhes de sua implementação (código) para que o objeto possa ser acessado. Para especificar essa interface dos objetos, a OMG utiliza a IDL (*Interface Definition Language*), como meio adotado para definir um padrão entre os objetos.

Com a definição das interfaces, o objeto cliente acessa um outro objeto pela execução de uma requisição ao mesmo, o qual tem suas características conhecidas por todos através de sua interface, que por sua vez é reconhecida por ser gerada a partir do padrão IDL, utilizado por todos os objetos da OMG. Em virtude disso, faz-se possível a interação entre diferentes objetos.

2.5 Arquitetura Cliente/Servidor

A arquitetura Cliente/Servidor vem sendo desenvolvida há vários anos. Primeiro, a realocação de aplicações em m*ainframe* para as chamadas plataformas abertas rodando o Sistema Operacional UNIX. Depois, com relação à abordagem dos dados, saindo de sistemas de arquivos ou banco de dados hierárquicos locados em *mainframes* para sistemas de banco de dados relacional e, posteriormente, a importância da capacidade gráfica dos pacotes de *front-end* existentes, facilitando a interação com o usuário [3,9].

Vários aspectos sobre uma definição da arquitetura Cliente/Servidor podem ser descritos, entre eles:

1) O termo Cliente/Servidor refere-se ao método de distribuição de aplicações computacionais através de muitas plataformas. Tipicamente essas aplicações estão divididas entre um provedor de acesso e uma central de dados e numerosos clientes contendo uma interface gráfica para usuários que necessitem acessar e manipular dados;
2) Cliente/Servidor geralmente refere-se a uma arquitetura onde dois ou mais computadores interagem de modo que um oferece os serviços aos outros. Este

modelo permite aos usuários acessarem informações e serviços de qualquer lugar;

3) Cliente/Servidor é uma arquitetura computacional que envolve requisições de serviços de clientes para servidores. Uma rede Cliente/Servidor é uma extensão lógica da programação modular.

Portanto, uma definição para a arquitetura Cliente/Servidor seria a existência de uma plataforma base para que as aplicações, onde um ou mais Clientes e um ou mais Servidores, juntamente com o Sistema Operacional e o Sistema Operacional de Rede, executem um processamento distribuído.

Assim, um sistema Cliente/Servidor poderia ser entendido como a interação entre s*oftware* e *hardware* em diferentes níveis, implicando na composição de diferentes computadores e aplicações.

Para melhor entender a arquitetura Cliente/Servidor é necessário observar que o conceito chave está na ligação lógica e não física. O Cliente e o Servidor podem coexistir ou não na mesma máquina [3]. Porém, um ponto importante para uma real abordagem Cliente/Servidor é a necessidade de que a arquitetura definida represente uma computação distribuída [3,7].

Um Cliente, também denominado de *front-end* ou *workstation* [9], é um processo que interage com o usuário através de uma interface gráfica ou não, permitindo consultas ou comandos para recuperação de dados e análise e representando o meio pelo qual os resultados são apresentados. Além disso, apresenta algumas características distintas:

1) É o processo ativo na relação Cliente/Servidor;
2) Inicia e termina as conversações com os Servidores, solicitando serviços distribuídos;
3) Não se comunica com outros Clientes;
4) Torna a rede transparente ao usuário.

Um Servidor, também denominado *back-end*, fornece um determinado serviço que fica disponível para todo Cliente que dele necessita. A natureza e escopo do serviço são definidos pelo objetivo da aplicação Cliente/Servidor. Além disso, ele apresenta ainda algumas propriedades distintas:

1) É o processo reativo na relação Cliente/Servidor;

2) Possui uma execução contínua;
3) Recebe e responde às solicitações dos Clientes;
4) Não se comunica com outros Servidores enquanto estiver fazendo o papel de Servidor;
5) Presta serviços distribuídos;
6) Atende a diversos Clientes simultaneamente.

O estilo de interação entre o usuário e o Cliente não precisa, necessariamente, ser feito por poderosas interfaces gráficas. Porém, já que o poder de processamento local do Cliente está disponível, pode-se aproveitá-lo integralmente, através de interfaces gráficas - GUI (*Graphical User Interface*), para melhor rendimento do usuário no seu trabalho.

Dentre as muitas vantagens da arquitetura Cliente/Servidor, pode-se citar: [9]

1) Confiabilidade - Se uma máquina apresenta algum problema, ainda que seja um dos Servidores, parte do Sistema continua ativo;
2) Matriz de Computadores agregando capacidade de processamento - A arquitetura Cliente / Servidor provê meios para que as tarefas sejam feitas sem a monopolização dos recursos. Usuários finais podem trabalhar localmente;
3) O Sistema cresce facilmente - Torna-se fácil modernizar o Sistema quando necessário;
4) O Cliente e o Servidor possuem ambientes operacionais individuais / Sistemas Abertos - Pode-se misturar várias plataformas para melhor atender às necessidades individuais de diversos setores e usuários.

Além destas vantagens, pode-se encontrar dentro de uma arquitetura Cliente/Servidor a interoperabilidade das estações Clientes e Servidoras entre as redes de computadores, a escalabilidade da arquitetura visando o crescimento e a redução dos elementos constituintes, a adaptabilidade de novas tecnologias desenvolvidas, a performance do hardware envolvido na arquitetura, a portabilidade entre as diversas estações que compõem a arquitetura e a segurança dos dados e processos.

Embora o avanço da arquitetura Cliente/Servidor tenha trazido uma variada gama de facilidades para o desenvolvimento de aplicações distribuídas, também possui algumas desvantagens:

1) Manutenção - As diversas partes envolvidas nem sempre funcionam bem juntas. Quando algum erro ocorre, existe uma extensa lista de itens a serem investigados;
2) Ferramentas - A escassez de ferramentas de suporte, não raras vezes obriga o desenvolvimento de ferramentas próprias. Em função do grande poderio das novas linguagens de programação, esta dificuldade está se tornando cada vez menor;
3) Treinamento - A diferença entre a filosofia de desenvolvimento de software para o microcomputador de um fabricante para o outro, não é como a de uma linguagem de programação para outra. Um treinamento mais efetivo torna-se necessário;
4) Gerenciamento - Aumento da complexidade do ambiente e a escassez de ferramentas de auxílio tornam difícil o gerenciamento da rede.

2.5.1 - Modelos da Arquitetura Cliente/Servidor

Existem vários modelos que podem ser utilizados na implantação da arquitetura Cliente/Servidor em sistemas de processamento distribuído [7]. A seguir serão descritos alguns deles de forma sucinta.

2.5.1.1 - Arquitetura Cliente/Servidor Simples

A primeira abordagem para um sistema distribuído é a arquitetura Cliente/Servidor Simples. Nesta arquitetura, o Servidor não pode iniciar nada. O Servidor somente executa as requisições do Cliente. Existe uma clara função de diferenciação, pois nesse modelo pode-se estabelecer que o Cliente é o mestre e o Servidor é o escravo.

2.5.1.2 - Arquitetura Cliente/Servidor em Dois Níveis

Nesta arquitetura existem vários Clientes requisitando serviços a um único Servidor. Ela caracteriza-se como sendo centrada no Servidor. Porém, na visão do usuário, existem vários Servidores conectados a somente um Cliente, ou seja, centrado no Cliente. Entretanto, com as várias possibilidades de comunicação possíveis, o que ocorre é uma mistura de Clientes e Servidores interagindo entre si, caracterizando uma comunicação mista, conforme pode ser verificado na FIGURA 06.

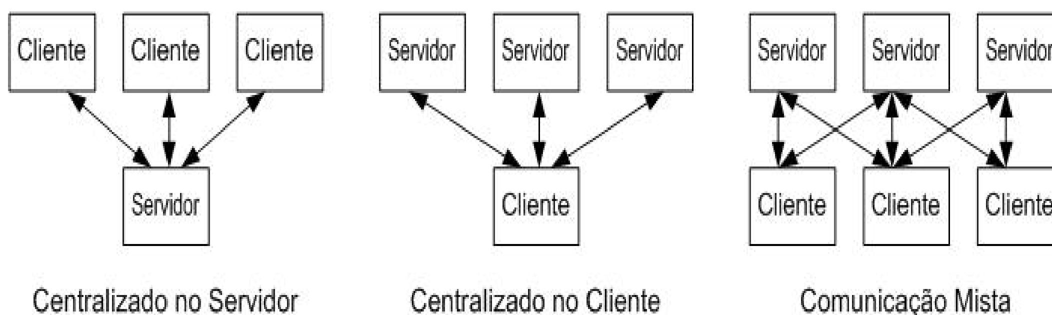


FIGURA 06 - Arquitetura Cliente/Servidor em 2 níveis

2.5.1.3 - Arquitetura Cliente/Servidor Multinível

Esta arquitetura permite que uma aplicação possa assumir tanto o perfil do Cliente como o do Servidor, em vários graus, ou seja, uma aplicação em alguma plataforma será um Servidor para alguns Clientes e, concorrentemente, um Cliente para alguns Servidores. Esta arquitetura pode ser vista na FIGURA 07.

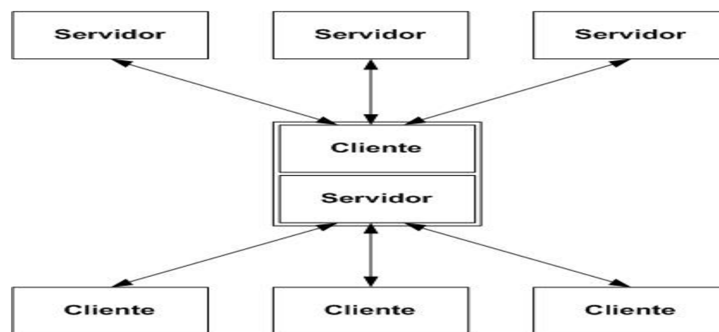


FIGURA 07 - Arquitetura Cliente/Servidor Multinível

2.5.1.4 - Arquitetura Cliente/Servidor Par-Par

Esta arquitetura pode ser vista como o caso mais geral da arquitetura Cliente/Servidor, ilustrado na FIGURA 08. Cada um dos pontos desta arquitetura assume tanto o papel de Cliente quanto de Servidor. Na verdade, torna-se pouco funcional lidar com quem é o Cliente ou o Servidor. É o caso onde o processo interage com outros processos em uma base pareada, não existindo nenhum Mestre ou Escravo; qualquer estação de trabalho pode iniciar um processamento, caso possua uma interface de comunicação entre o usuário e o processo Cliente.

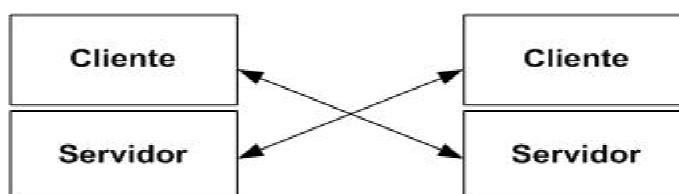


FIGURA 08 - Arquitetura Cliente/Servidor Par-Par

2.5.2 – Processos em uma Arquitetura Cliente/Servidor

A arquitetura Cliente/Servidor divide uma aplicação em processos que são executados em diferentes máquinas conectadas à uma Rede de Computadores, formando um único sistema. O paradigma da tecnologia

Cliente/Servidor serve como um modelo, entre outros para interação entre processos de softwares em execução concorrente [5].

Os processos ou tarefas a serem executadas são divididas entre o Servidor e o Cliente, dependendo da aplicação envolvida e das restrições impostas pelo SOR (Sistema Operacional de Rede). Quanto mais avançado for o Sistema Operacional de Rede, menor será a aplicação em si, uma vez que a implementação do código para acessar a rede já se encontra no SOR.

Atualmente dois tipos de processamentos estão sendo divulgados: Processamento Distribuído e Processamento Cooperativo. A característica de cada um destes é descrita a seguir [7]:

1) Processamento Distribuído - A distribuição de aplicações e tarefas se faz através de múltiplas plataformas de processamento. O processamento distribuído implica que essas aplicações ou tarefas irão ocorrer em mais de um processo, na ordem de uma transação a ser concluída. Em outras palavras, o processamento é distribuído entre duas ou mais máquinas e os processos, na maioria, não rodam ao mesmo tempo. Por exemplo, cada processador realiza parte de uma aplicação em uma seqüência. Geralmente, o dado usado em um ambiente de processamento distribuído também é distribuído através de plataformas;
2) Processamento Cooperativo - A cooperação requer dois ou mais processadores distintos para completar uma simples transação. O processamento cooperativo é relatado para ambos os processos, cliente ou servidor. É uma forma de computação distribuída onde dois ou mais processadores distintos são requeridos para completar uma simples transação de negócios. Normalmente esses programas interagem e executam concorrentemente como processos diferentes. Os processos cooperativos também são considerados como um estilo de Cliente/Servidor através da arquitetura de mensagens, que devem obedecer a um determinado padrão.

No contexto do presente trabalho pretende-se utilizar as características do processamento distribuído, que poderá ser visto com mais detalhes no próximo tópico.

2.5.2.1 - Processamento Distribuído

O processamento distribuído, também denominado de processamento concorrente, utiliza-se do mecanismo de passagem de mensagens para a comunicação entre processos, que podem ser de quatro tipos básicos: Filtro, Ponto-a-Ponto, Cliente e Servidor [7,17].

Os filtros atuam no processo de transferência como conversores de mensagens de comunicação entre o usuário e o *host*. Ex.: Ligação de um *desktop* com um *mainframe* através de um emulador de terminal.

Na comunicação Ponto-a-Ponto, existem os processos duplicados executando em todas as máquinas e prestando serviços uns para os outros. Não existem processos servidores estabelecendo um Servidor dedicado; cada processo pode ser Cliente e Servidor para outros processos.

O modelo que utiliza processos Cliente/Servidor depende do cliente para inicializar a comunicação. Sua característica básica é que processos Clientes enviam pedidos a um processo Servidor, que retorna o resultado para o Cliente. O processo Cliente fica então liberado da ação do processamento da transação podendo realizar outros trabalhos.

2.5.3 Camadas da Arquitetura Cliente / Servidor

A arquitetura Cliente/Servidor é dividida em três camadas básicas: Aplicação, Serviços do Sistema e *Hardware*. A camada de Aplicação compõe-se dos processos da aplicação, entre eles, os processos Cliente e Servidor. A camada de Serviços de Sistemas compreende o Sistema Operacional e o Sistema Operacional de Rede, destinando-se ao controle do hardware. Por último, a camada de *hardware*, onde estão localizados os periféricos conectados aos Clientes e Servidores.

A tecnologia Cliente/Servidor pode existir tanto no nível da camada de Aplicação, quanto no da camada de Serviços do Sistema. A coexistência do paradigma nestas camadas surge em função da hierarquia das atuações no sistema. Caso o usuário seja externo ao sistema, os processos Cliente e Servidor compõem a camada da Aplicação; se o usuário for um programa de

aplicação o Cliente é um processo redirecionador e o Servidor será um processo respondedor da camada de Serviços do Sistemas.

Para sistemas Cliente/Servidor na camada de Aplicação, a camada Serviços do Sistema oferece somente um mecanismo de IPC (*InterProcess Comunication*) para troca de mensagens. Por outro lado, a camada Serviços do Sistema configurada como Cliente/Servidor, é responsável por gerenciar o redirecionamento das solicitações de gravação ou leitura, por exemplo. É importante notar que a diferença entre os sistemas Cliente/Servidor nas camadas de Aplicação e Serviços do Sistema é o equilíbrio entre a quantidade de processamento, tanto no lado do Cliente, quanto no lado do Servidor.

Existem vários sistemas que podem ser baseados na estrutura Cliente/Servidor. O uso mais freqüente são as aplicações de Banco de Dados usando processos *SQL* (*Structured Query Language*) de *front-end*, para acessar remotamente, as bases de dados.

A FIGURA 09 mostra uma estrutura baseada num Servidor de Arquivos. Esta estrutura ocasiona um maior fluxo de informações na rede, uma vez que todo o arquivo será transferido para o Cliente para então ser trabalhado.

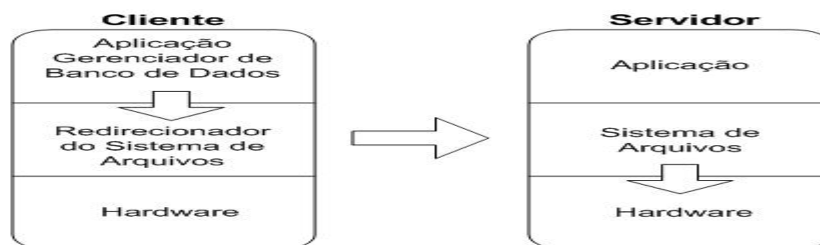


FIGURA 09 - Arquitetura Cliente/Servidor como Servidor de Arquivo

Nesta estrutura, a camada de Aplicação passa a ser o Cliente do Sistema. Com isto, a camada de Serviço do Sistema é utilizada simplesmente como um redirecionador para acesso à base de dados. Neste tipo de configuração pode-se chamar este Sistema como falso Sistema Cliente/Servidor, por não haver um equilíbrio de processamento entre os dois lados. O lado Servidor somente terá o trabalho de executar as rotinas comuns de entrada e saída, não sendo caracterizado como um processamento intrínseco à aplicação.

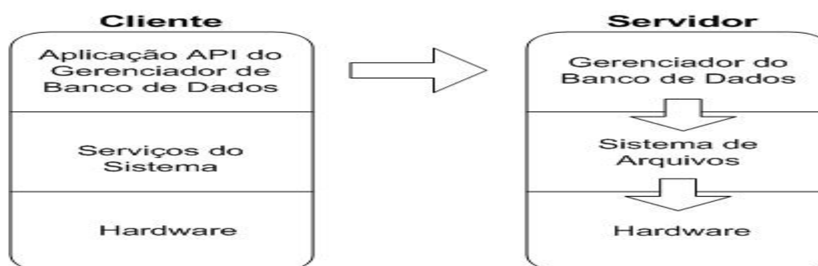


FIGURA 10 - Arquitetura Cliente / Servidor como Servidor de Banco de Dados

A FIGURA 10 demonstra outra possibilidade de se estruturar um sistema baseado na arquitetura Cliente/Servidor, que consiste na utilização de um Servidor de Banco de Dados dedicado, podendo coexistir normalmente com o Servidor de arquivos. Neste momento, com um Servidor de Banco de Dados exclusivo, o fluxo de informações trafegadas na rede diminui, já que somente a resposta da consulta é retornada ao Cliente, ao invés de transferir todo o arquivo como era feito anteriormente.

O Cliente, neste tipo de estrutura, é o usuário. Passa-se a ter uma visão da aplicação como se as partes, Cliente e Servidor, fossem algo único. A FIGURA 11 exemplifica esta visão.

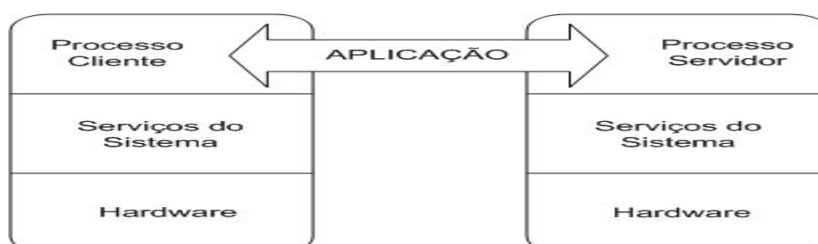


FIGURA 11 - Integração entre os processos clientes e servidores

Este tipo de estrutura favorece o aumento da performance da rede de comunicação, possibilitando assim, um maior número de ligações simultâneas de diversos Clientes com diversos Servidores.

Embora a arquitetura Cliente/Servidor possa parecer uma nova versão do modelo de arquivos compartilhados através da Rede Local baseada em microcomputadores, suas vantagens são inúmeras. Fundamentalmente, ambos os modelos provêem capacidade de processamento distribuído e permitem o

compartilhamento de informação. Entretanto, no Modelo de Arquivos compartilhados baseado em Servidor de Arquivos, o Cliente além de executar a aplicação, executa também o motor da Base de Dados, que por sua vez acessa essas Bases de Dados remotamente como se fossem locais. O Servidor de Arquivos envia arquivos inteiros através da rede para o Cliente processar localmente, ocasionando congestionamento na rede. Já no modelo Cliente/Servidor, o Cliente executa parte da aplicação, sendo deixado para o Servidor a tarefa da administração da Base de Dados, comumente exercida por algum SGBD. O Servidor envia para o Cliente, através da Rede, apenas o bloco de dados apropriado, como resultado da consulta efetuada pela aplicação.

Desta forma, na arquitetura Cliente/Servidor, cada Servidor pode suportar um número maior de usuários, uma vez que é o Cliente que gerencia a aplicação e a interface com o usuário. Além do mais, com a crescente conectividade entre máquinas e sistemas operacionais, pode-se escolher para Cliente o ambiente de software e hardware que melhor se adequar às necessidades de cada aplicação do usuário, sem ter que se preocupar com o Servidor.

A melhor divisão de tarefas entre o Cliente e o Servidor depende de cada aplicação em si. Se o Servidor for apenas um Sistema Gerenciador de Banco de Dados, deixando para o Cliente o resto do processamento, ou se algumas tarefas como controle de acesso, análise dos dados e validação dos comandos são executadas pelo Servidor, é uma decisão do construtor do Sistema em função das características do negócio do usuário.

2.5.4 IPC - Interprocess Communication

Um processo pode ser entendido como um programa em execução. Considerando que vários processos podem estar em execução e, freqüentemente, compartilham recursos, faz-se necessário o uso de mecanismos que possibilitem e coordenem a comunicação e sincronização entre os mesmos [7,14,17].

A comunicação permite que processos que interagem na resolução de determinada aplicação troquem informações. A sincronização provê controle

de acesso e controle de seqüência para garantir a consistência na comunicação entre processos.

Os processos podem estar em execução no mesmo computador ou em diferentes computadores conectados através de uma rede. Por isso existem mecanismos de IPC que permitem comunicação entre processos remotos, tal como o RPC e *Socket*, comentados anteriormente.

A comunicação entre processos é requerida em todos os sistemas operacionais multitarefa ou multiprocesso e, geralmente, não é suportada por sistemas operacionais monotarefa ou monoprocessados como o DOS. O OS/2 e o Microsoft Windows suportam um mecanismo IPC chamado DDE (*Dynamic Data Exchange*), que consiste na troca dinâmica de dados, baseado no conceito Cliente/Servidor.

2.5.4.1 Mecanismos de IPC

Os mecanismos de IPC têm a característica de implementarem de maneira distinta, a comunicação através da passagem de mensagem, *mail-box e* conexões *multicast* bem como o sincronismo entre processos.

Dentre esses mecanismos de sincronização destacamos: semáforos, troca de mensagens e filas de mensagens.

2.5.4.1.1 Semáforos

Um semáforo é uma variável compartilhada inteira e não negativa que indica o estado do recurso ao qual esta associada. É um mecanismo de IPC que pode implementar controle de acesso e controle de seqüência [5,7]. Um semáforo só pode ser manipulado por duas operações, que são:

1) *down* - implementada como: *down*(semáforo)
   se semáforo=0 então processo_fica_em_espera senão semáforo=semáforo-1;
2) *up* - implementada como: *up*(semáforo)
   semáforo=semáforo+1.

As primitivas acima são ações indivisíveis. Isso garante que quando um processo inicia uma das operações sobre um semáforo, nenhum outro processo terá acesso a ele até que a operação se conclua.

Em um semáforo que está sendo utilizado para controle de acesso, o valor 0 (zero) indica que o recurso está sendo utilizado por outro processo, o valor 1 indica que o recurso está disponível. Em um semáforo que está sendo utilizado para controle de seqüência, o valor 0 (zero) indica que não há nenhum processo em espera, um valor maior que 0 (zero) indica a quantidade de processos em espera.

Quando um processo deseja usar um recurso compartilhado, ele executa, sobre determinado semáforo, uma operação *down*. Esta operação verifica o valor do semáforo e se o valor for 0 (zero) ela coloca o processo solicitante em espera, senão o processo solicitante continua em execução. Quando o processo não precisar mais do recurso, ele deve executar uma operação *up* sobre o semáforo determinado, liberando o recurso para outros processos.

2.5.4.1.2. Troca de mensagens

É um método para comunicação entre processos, no qual, processos enviam e recebem mensagens ao invés de compartilhar variáveis. As operações de troca de mensagens são generalizadas pelo uso de primitivas *send/receive* (envia/recebe), cujas sintaxes podem ser:

1) *send* mensagem *to* processo_destino;
2) *receive* mensagem *from* processo_origem.

Essas operações podem ser bloqueante ou síncrona ou não bloqueante ou assíncrona. No primeiro caso, o processo que envia a mensagem fica bloqueado até que receba uma confirmação de recebimento da mensagem do processo receptor. No segundo caso, o processo que envia a mensagem não necessita esperar confirmação, mas neste caso a mensagem deve ser armazenada em um *buffer*. As primitivas *send/receive* fornecem subsídios para a construção de algumas abstrações de comunicação, tal como o RPC e *Socket*.

2.5.4.1.3. Filas de mensagens

Permitem que, através do uso de funções de manipulação de filas de mensagem, os processos mandem listas formatadas de dados para qualquer processo. Existem quatro chamadas para a troca de mensagens:

1) *msgget()* - retorna um descritor de mensagens que designa a lista de mensagens a ser utilizada;
2) *msgctl()* - possui uma opção que seta e retorna parâmetros associados com o descritor de mensagens e uma opção que remove o descritor;
3) *msgsnd()* - manda a mensagem desejada;
4) *msgrcv()* - recebe a mensagem.

Quando é usado o *msgget()* para criar um novo descritor, o sistema operacional procura o vetor de filas de mensagens para verificar se existe uma com a chave dada. Caso não tenha esta entrada, o sistema operacional aloca uma nova fila na estrutura, inicializa e retorna um identificador para o usuário; caso contrário, verifica a permissão e retorna o identificador.

Um processo usa o *msgsnd()* para mandar uma mensagem. Esta chama um descritor para a lista de mensagens, um ponteiro para a estrutura, um contador com o tamanho do vetor de dados e um *flag* que indica para o sistema operacional se deve executar fora do espaço do *buffer* interno. O sistema operacional aloca espaço para a mensagem no mapa de mensagens e copia os dados para o espaço do usuário. Ele aloca o cabeçalho da mensagem e coloca no fim da lista encadeada de cabeçalhos de mensagens, grava o tipo e tamanho da mensagem no cabeçalho, direciona o apontador para a mensagem. Logo em seguida, o sistema operacional dispara o processo que estava esperando pela mensagem.

Um processo recebe uma mensagem pela chamada *msgrcv()* que contém o endereço da estrutura do usuário, onde está a mensagem e o tamanho da mesma. O sistema operacional verifica se o usuário tem direito de acesso; se possuir aloca a primeira mensagem da lista encadeada ou, se especificado, a primeira mensagem do tipo solicitado. Se o tamanho é menor ou igual ao tamanho requisitado pelo usuário, o sistema operacional copia a mensagem para a estrutura

de usuário, decrementando o contador de mensagens da lista. O sistema operacional ainda armazena o tempo de transferência, a identificação do processo que recebeu a mensagem, ajusta os apontadores da lista encadeada e libera a área do sistema operacional que tinha armazenado a mensagem.

2.5.4.1.4 Sockets

Na maioria dos sistemas modernos, a comunicação entre máquinas ou em processos residentes em uma mesma máquina, é realizada através do uso de *sockets*. O *socket* é a base para a transmissão e recepção em uma rede [3]. Ele é a generalização do mecanismo de acesso a dispositivos, fornecendo um ponto terminal para a comunicação na maioria dos sistemas operacionais utilizados, como Unix, Linux e Windows. Assim como no acesso a arquivos, os programas aplicativos solicitam ao sistema operacional a criação de um *socket* quando necessário. O sistema retorna um identificador como sendo um número inteiro de valor baixo que o aplicativo utilizará como referencia ao *socket* recém criado. Diferente do descritor de arquivo, que vincula a operação a um dispositivo específico quando o processo chama a primitiva *abrir*, o *socket* pode ser criado sem estar vinculado a endereços de destino específicos. O aplicativo pode optar, dependendo das necessidades, por fornecer um endereço de destino a cada vez que usar o *socket*, como ocorre nos protocolos de transporte não-orientados à conexão no envio de datagramas, ou então, optar pelo endereçamento estático, como ocorre com os protocolos de transporte orientados a conexão. Para tornar confiável o acesso às primitivas de *ler* e *gravar* pelo sistema operacional, o identificador numérico dos descritores de arquivo e *sockets* serão sempre diferentes para o aplicativo [3].

Existem três tipos de *sockets* [14]:

1) *Datagram Sockets* – Envia o pacote sem a necessidade de confirmação pelo destino. No TCP/IP esta interface é implementada pelo protocolo *UDP – User Datagram Protocol*;
2) *Stream Sockets* – Antes da transmissão do pacote, uma conexão ponto a ponto obrigatoriamente deverá estar efetivada, sendo que durante o processo de

transferência, o destinatário terá a garantia da recepção. No TCP/IP esta interface é implementada pelo protocolo *TCP – Transfer Control Protocol*;

3) *Raw Sockets* – Envia o pacote sem utilizar as camadas de transporte e sem os serviços de uma conexão ponto a ponto tradicional. No TCP/IP esta interface é implementada na camada de rede, a partir do protocolo *ICMP – Control Mensage Protocol*.

2.5.4.2 Comunicação entre Processos de Aplicações Cliente / Servidor

A comunicação entre Cliente e Servidor procede de forma implícita. Quando o Cliente espera a resposta da mensagem enviada para continuar o seu processamento, diz-se que o protocolo utilizado é um protocolo com *bloqueio*, onde o sincronismo entre Cliente e Servidor está implícito no mecanismo de passagem de mensagem [7].

Caso o Cliente possa continuar suas tarefas, enquanto espera a resposta da mensagem, o protocolo de comunicação é um protocolo *sem bloqueio*. Isto ocorre quando o sistema operacional do Cliente é multitarefa ou multiprocessamento, possibilitando ao Cliente executar outras tarefas enquanto aguarda a resposta do Servidor.

A teoria de programação concorrente é baseada na noção de processos de comunicação sendo executados em paralelo a outros processos. Esses processos se comunicam compartilhando memória ou passando mensagens por meio de um canal de comunicação compartilhado. O termo IPC (*Interprocess Comunication*) se refere às técnicas utilizadas na passagem de mensagem.

No compartilhamento de memória, os processos concorrentes compartilham uma ou mais variáveis, e utilizam a mudança de estados dessas variáveis para se comunicarem. Essa técnica inclui espera ocupada, semáforos, regiões críticas condicionais e monitores [7]. Como esta técnica exige que os processos estejam na mesma máquina, não são considerados base para a programação Cliente/Servidor.

Em técnicas baseadas na passagem de mensagem, os processos enviam e recebem mensagens explicitamente, em vez de examinar o estado de

uma variável compartilhada. O benefício principal da passagem de mensagem é que existe pouca diferença entre o envio de mensagens a processos remotos ou locais. Portanto, a passagem de mensagem é poderosa para criação de aplicações em rede. Outra vantagem é que mais informações podem ser trocadas numa mensagem do que por mudança no estado de uma variável compartilhada.

Quando se deseja desenvolver e utilizar aplicações distribuídas, existem quatro alternativas básicas que podem ser utilizadas para a implementação da comunicação entre os processos, ficando a escolha sob a responsabilidade do projetista, que deverá selecioná-la a partir da natureza da aplicação e os recursos envolvidos como sistema operacional, linguagem de programação entre outros. Estas alternativas são RPC, COM/DCOM, CORBA e SOCKETS, descritos anteriormente.

2.6 Monitoração da Comunicação entre Aplicações Cliente/Servidor

Aplicações desenvolvidas segundo as definições cliente/servidor nem sempre se comportam de acordo como foram projetadas. Isto pode ocorrer por falhas no projeto original, erros de implementação ou então por não estar atendendo a algum critério estabelecido pela estrutura de rede. Uma sistemática de monitoração e depuração tem como finalidade básica auxiliar o projetista a identificar os problemas de execução citados acima, e também possibilitar um acompanhamento dos procedimentos e tarefas da aplicação.

A monitoração e a depuração poderão ocorrer a partir da pilha de protocolos utilizados ou a partir da Interface de comunicação entre o protocolo e o aplicativo, que no caso deste estudo, será fundamentado no mecanismo de interação denominado *Socket,* visto anteriormente.

No caso da pilha de protocolos, existem ferramentas enquadradas na categoria NAM (*Network Animator*) que, através de editores gráficos de animação e simulação, possibilitam um cenário capaz de facilitar o projeto e a monitoração de um protocolo [18]. O mecanismo utilizado é a captação de uma grande quantidade de informações em um curto período de tempo e com elas simular e acompanhar as condições de tráfego.

A monitoração de uma aplicação utilizando *Socket*, neste estudo a API W*insock,* baseado no sistema operacional Windows, pode ocorrer segundo duas possibilidades, que são [14,19]:

1) A API *winsock* padrão do Windows pode ser substituída por outra API contendo o mesmo conjunto de chamadas aos protocolos do sistema, com a diferença que, além do conjunto de argumentos padronizados, outros foram adicionados com funções específicas de monitoração, para serem utilizadas a partir de uma aplicação desenvolvida com esta finalidade;
2) Implementação de uma ferramenta de monitoração e depuração que utiliza a interface de monitoração da Microsoft, acessando as funções disponíveis na API *Winsock* que, através de um mecanismo de filtros para captura de pacotes, mais especificamente de datagramas IP, disponibilize uma série de funcionalidades que permitem a visualização, controle e interação no processo de transferência de dados entre os processos. Esta técnica é a mais compatível com as diversas versões do Windows, por utilizar os recursos oferecidos por este sistema operacional.

Uma ferramenta eficiente de monitoração deve ser capaz de reconhecer qualquer tipo de chamada às primitivas residentes na API, identificar seus argumentos e seus valores durante a execução, bem como relacionar através da aplicação os processos em execução, a interação entre eles e quais protocolos estão em atividade no momento da verificação.

2.6.1 SNIFFERS - Ferramentas para Captura de Pacotes

Os *Sniffers* têm como objetivo principal, a captura e ou análise de pacotes em uma rede, sendo esta local ou de longa distância. Analisa todo o tráfego de uma rede, seja verificando, ou apenas contabilizando os tipos de pacotes que estão trafegando (TCP, ICMP, UDP, etc).

A primeira finalidade pode ser encarada como possível ataque, pois existe a interceptação do pacote que está sendo transmitido e este, caso não esteja criptografado, pode ser facilmente interpretado pelo *software*. A segunda já é de caráter estrutural e visa fornecer uma visão detalhada do tráfego em uma

rede. Tem por objetivo fornecer embasamento para decisões técnicas que possam vir a melhorar o desempenho da infra-estrutura física ou até mesmo de s*oftwares* que utilizem comunicação em rede, cliente-servidor, por exemplo.

A técnica utilizada para esta monitoração é bastante simples. Na grande maioria das redes, os pacotes são transmitidos para todos os computadores conectados ao mesmo meio físico, sendo que cada máquina é programada para somente tratar os pacotes destinados para ela. Entretanto, é possível reprogramar a interface de rede de uma máquina para que ela capture todos os pacotes que circulam pelo meio, não importando o destino. Este modo de operação da interface de rede é denominado modo promíscuo e a técnica é denominada *sniffing*.

As informações e a forma como são capturadas dependem muito do *software* que está sendo utilizado. Entretanto, em geral as informações capturadas são (quando utilizado por invasores): nome do *host* de destino, *username* e *password*. A informação é gravada em um arquivo posteriormente recuperado pelo invasor que acessa outras máquinas.

Normalmente o volume de dados a tratar é muito grande, tarefa que pode ser extremamente facilitada através de técnicas de filtragem e pelo fato de ser extremamente fácil detectar o início de uma conexão TCP. Essas técnicas de filtragem de pacotes podem ser bem empregadas quando se fala de análise de desempenho em uma rede onde se pode medir a quantidade de pacotes de um determinado tipo que estejam trafegando no meio físico. Por exemplo, caso existam muitas solicitações de reenvio de pacotes, pode-se concluir que existe algum problema no meio físico ou roteamento de pacotes na rede. Dessa forma, podemos deduzir que quando se está analisando o tráfego na rede o que interessa basicamente é o cabeçalho do pacote, enquanto que quando o *software* está sendo utilizado para ataques, o cabeçalho é importante apenas para determinar qual é o tipo do pacote e retirar as informações desejadas (filtragem).

2.7 Resumo do Capítulo

Neste capítulo foram revistos alguns conceitos que serão de vital importância no desenvolvimento deste trabalho. Inicialmente foram abordados alguns tópicos relacionados a redes de computadores, à padronização de protocolos e em especial, uma breve introdução à Família de Protocolos TCP/IP. Em seguida, buscou-se, com maior relevância, conceituar o paradigma Cliente/Servidor, sua arquitetura básica, a distribuição de processos e os mecanismos utilizados na comunicação entre eles. Como isso, chegou-se a um parâmetro para a modelagem do Ambiente para Geração de Aplicações Distribuídas proposto neste projeto, que será baseado em uma plataforma composta pelo Sistema Operacional Windows da Microsoft, com o protocolo TCP/IP e utilizando a interface de comunicação via *sockets* entre os processos das aplicações baseadas no modelo Cliente/Servidor. A opção por este mecanismo de comunicação deve-se à diminuição da latência do envio e recebimento de mensagens na rede proporcionadas por esta abordagem, o que é particularmente importante em sistemas de tempo real [3,8,13], já que um dos objetivos deste trabalho será a integração do ambiente proposto com um gerador de programas paralelos de tempo real denominado TEV (*Teaching Environmento for Virtuoso*).

Nos próximos capítulos, serão demonstrados a modelagem do Ambiente, os recursos utilizados e a interação deste com o TEV.

Capítulo 3
API NETWORK

3.1 Modelagem e Implementação da API Network

Neste capítulo será descrito a modelagem e implementação de uma biblioteca de funções denominada *Network*, baseada na interface *Socket*, abstração utilizada na comunicação entre aplicativos e os protocolos de uma rede de computadores. Esta biblioteca contém um conjunto de primitivas que implementam o acesso aos serviços dos protocolos para serem utilizados na comunicação em rede pelos processos que forem especificados a partir do AGAD (Ambiente para Geração de Aplicações Distribuídas), detalhada no próximo capítulo. A implementação foi realizada com a utilização da linguagem C, a partir do Ambiente de Desenvolvimento Visual C++ 6.0 [20].

Os componentes desenvolvidos incorporam algumas funcionalidades que possibilitam a implementação de aplicações distribuídas em uma rede através da utilização das interfaces de *sockets*, e ainda proporcionam aos projetistas meios que permitem a monitoração e depuração das etapas pertinentes aos serviços de comunicação. Isso é possível devido à modelagem dos componentes em três grupos principais. O primeiro será responsável pela construção de aplicações Servidor; o segundo pela construção de aplicações Cliente, através da utilização de funções que interagem com as primitivas da Interface *Sockets*; o terceiro proporciona ao usuário funcionalidades para que o mesmo possa verificar, acompanhar e analisar o tráfego existente na rede entre as aplicações especificadas com o Ambiente, tornando o processo de comunicação mais transparente durante as etapas de construção.

A seguir serão descritos como ocorre a comunicação entre uma aplicação e um protocolo de rede a partir da Interface *Socket*, a modelagem de uma biblioteca para a utilização de suas primitivas e, em seguida, a construção e utilização de um Ambiente para ser utilizado na elaboração e monitoração de aplicativos.

3.2 Descrevendo a Interação entre Processos através dos Protocolos

Embora conexões de redes e protocolos de comunicação sejam necessários para a comunicação através de uma rede, a funcionalidade mais interessante e útil é fornecida pelo programa aplicativo. Eles fornecem o serviço de alto nível que os usuários acessam e determinam como os mesmos percebem as capacidades da rede subjacente. Os aplicativos determinam o formato em que as informações são exibidas e os mecanismos que os usuários têm para selecionar ou acessar estas informações.

O aplicativo não acessa o protocolo de comunicação diretamente. Isto ocorre através de uma interface que serve como elo de ligação entre os dois. Esta interface não segue um padrão específico, como ocorre na implementação de um protocolo. Os padrões não especificam exatamente como os programas aplicativos interagem com a estrutura do protocolo. No entanto, levando-se em consideração que o protocolo está vinculado à implementação do sistema operacional, tem-se que os detalhes da interface devem obrigatoriamente seguir as especificações do sistema operacional.

Embora um sistema de redes forneça um serviço de comunicação básico, o software de protocolo não pode iniciar ou aceitar contato de um computador remoto. Em vez disso, dois programas aplicativos devem participar em cooperação mútua no processo de interação por intermédio do paradigma Cliente/Servidor, onde o Cliente inicia a comunicação ativamente e o Servidor espera passivamente por um contato.

A interface utilizada pelos processos de aplicações Cliente/Servidor, quando interagem com o protocolo de transporte, é conhecida como API (*Application Programming Interface)* [3,14,19]. Uma API define um conjunto de operações que servirão como mecanismos que irão implementar as funcionalidades na criação deste tipo de aplicativos. Como já mencionado, normalmente, os padrões de protocolo não especificam uma API para ser utilizada pelos aplicativos para se interagirem. Em vez disso, os protocolos especificam as operações gerais que devem ser fornecidas e permitem que cada sistema operacional defina a API específica, de acordo com suas particularidades. Estas operações

poderiam ser resumidas em *abrir-ler-escrever-fechar*. Com estas primitivas básicas, um processo de usuário que necessitar de alguma operação de entrada e saída, deverá inicialmente executar uma chamada à primitiva *abrir*, especificando um arquivo ou dispositivo a ser utilizado, obtendo com isto a permissão para a operação desejada. A primitiva *abrir* irá retornar um descritor de arquivos de número inteiro, o qual será utilizado pelo processo, servindo como identificador para as operações de entrada e saída. Uma vez identificado, o processo poderá executar chamadas às primitivas *ler* ou *gravar* para a transferência de dados. Estas duas primitivas utilizam três parâmetros que especificam o descritor de arquivos a ser usado, o endereço do *buffer* e o número de *bytes* a ser transferido. Após a conclusão das operações de transferência, o processo do usuário faz uma chamada à primitiva *fechar*, informando ao sistema operacional para executar as funções necessárias à desativação de todos os descritores de arquivo que estiverem sendo utilizados.

3.3 Programando Sockets

A seguir, serão detalhadas as principais funções necessárias para a utilização de *sockets*.

3.3.1 Criação de um *Socket*

A primitiva *socket()* será utilizada por um aplicativo cliente/servidor na criação de um *socket*, sendo necessários três argumentos inteiros: *descritor = socket(fp,tipo,protocolo).*

Em *fp* será especificada a família de protocolos a ser usada com o *socket.* O valor deste argumento poderá ser *PF_INET* quando em redes TCP/IP ou *PF_UNIX* quando em redes Unix.

O argumento *tipo* especifica o tipo de comunicação desejada, podendo ser *SOCK_STREAM,* quando houver necessidade de uma comunicação confiável, ou então *SOCK_DGRAM*, quando o serviço exigir uma comunicação não orientada à conexão. O argumento *protocolo* especifica qual o protocolo da família escolhida que

será utilizado pelo *socket*. Isto é necessário em função da compatibilização entre este protocolo e o tipo da conexão utilizada.

O resultado do acesso à primitiva *socket* será retornado como um inteiro armazenado na variável *descritor*.

3.3.2 Fechamento de um Socket

Quando um processo finalizar a utilização de um *socket*, será invocada a primitiva *close* como segue: *close(descritor).*

O argumento *descritor* é a identificação do *socket* a ser fechado. Caso o aplicativo não encerre normalmente o *socket* por qualquer motivo, o sistema operacional executará esta operação por segurança.

3.3.3 Especificação de um Endereço Local

Quando um *socket* é criado, a identificação da origem ou destino não é especificada. No caso dos protocolos TCP/IP, isto significa que nenhuma porta foi alocada. No entanto, para a ligação entre um processo de um aplicativo servidor e um processo de um aplicativo cliente, necessita-se, em muitos casos, dos endereços de identificação. Esta tarefa é de responsabilidade da aplicação servidor, através da chamada da primitiva *bind(descritor, endloc, tend)*.

O argumento *descritor* é a identificação do *socket* a ser vinculado. O argumento *endloc* é a estrutura que especifica o endereço local ao qual o *socket* deve ser vinculado e o argumento *tend* especifica o comprimento em *bytes* do endereço.

Por razões de simplificação do processo de vinculação, o argumento *endloc* é definido como uma estrutura de dados que identifica detalhadamente a conexão local através da seguinte forma: *endloc = struct {FAMÍLIA_ENDEREÇOS, PORTA, IP}*. O argumento *FAMILIA_ENDEREÇOS* contém 16 bits que identificam a pilha de protocolos à qual a família de endereços pertence. Por exemplo, se seu valor for igual a dois, significa que o endereçamento segue os padrões da pilha de protocolos TCP/IP. O argumento *PORTA* identifica o número da porta do protocolo (quando TCP/IP) a ser vinculada e, por fim, o argumento *IP*

identifica o endereço local de acordo com as especificações da família de protocolo.

3.3.4 Especificação de um Comprimento de Fila para um Servidor

Existem algumas situações em que um aplicativo servidor deve entregar dados aos aplicativos clientes de forma confiável. Isto é possível através de um mecanismo de armazenamento das solicitações em fila até o momento em que forem processadas. A chamada de sistema *listen* permite que um servidor prepare um *socket* para conexões de entrada. O *socket* será ativado em forma passiva, pronto para aceitar novas conexões. A forma de utilização da primitiva é a seguinte: *listen(descritor, tf).* O argumento *tf* especifica o comprimento da fila de solicitação para o *socket* definido no argumento *descritor*.

3.3.5 Preparação do Servidor para Aceitar Conexões

Como pode ser observado, um processo do servidor utiliza as chamadas de sistema *socket, bind* e *listen* para a criação, vinculação e definição da fila de solicitações de um novo *socket*. A primitiva *bind* apenas associa o *socket* a uma porta de protocolo conhecida, mas o *socket* não é conectado a um destino externo específico. Na realidade, um servidor dever ser projetado de forma a aceitar diversas solicitações de um cliente arbitrário.

A partir do momento em que um *socket* servidor é estabelecido, o mesmo ficará em estado bloqueado, aguardando uma conexão. A primitiva responsável por esta tarefa é a chamada de sistema denominada *accept,* utilizada da seguinte forma: *novosocket=accept(descritor, endereço, tend)*. O argumento *endereço* é uma estrutura do tipo *endloc.*

3.3.6 Vinculação de Sockets a Endereços Destino

Inicialmente o *socket* é criado no estado desconectado. Para alterar este estado para conectado e então permitir a transferência através de um fluxo

confiável, será necessário a utilização da primitiva *connect* da seguinte forma: *connect(descritor, destino, tend).*

O argumento *descritor* identifica o *socket* a ser conectado. O argumento *destino* é similar ao argumento *endloc* da primitiva *bind* e o argumento *tend* determina o tamanho em *bytes* do endereço destino.

3.3.7 Transmissão de Dados Através de um Socket

A transferência de dados através de um *socket* poderá ocorrer através de cinco primitivas básicas (*send, sendto, sendmsg, write e writev*). *Send, write e writev* não necessitam do endereço destino, no entanto, só funcionam se o *socket* já estiver conectado.

A primitiva *write* deverá ser utilizada da seguinte forma: *write(descritor, buffer, tamanho).* O argumento *buffer* especifica o endereço dos dados a serem enviados e *tamanho* especifica o número de bytes a serem enviados.

A primitiva *writev* é similar à primitiva *write*, porém, ela possibilita ao protocolo escrever uma mensagem sem copiá-la em bytes contíguos na memória. Isto ocorre por que existe um argumento denominado *iovector* que contém a seqüência de ponteiros para os blocos que formam a mensagem, conforme demonstrado na sintaxe *writev(descritor, iovector, tamvetor).* O argumento *tamvetor* especifica o número de entradas em *iovector.*

O envio de dados poderá ocorrer a partir das primitivas *send* e *sendto*. Quando a transmissão for a partir da primitiva *send*, a mesma deverá ocorrer através da seguinte forma: *send(descritor, mensagem, tamanho, controle).* O argumento *mensagem* fornece o endereço dos dados a serem enviados, *tamanho* especifica o número de bytes e *controle* pode ser utilizado como um mecanismo de acompanhamento da transmissão. Este último argumento é importante para o aplicativo porque nele poderão estar definidos alguns critérios a serem observados pela estrutura de rede. Por exemplo, no caso de dados urgentes do TCP/IP, o aplicativo poderá optar pela transmissão fora de banda, ou então, permitir ao projetista definir como irá ocorrer o roteamento da mensagem.

Quando houver a necessidade da especificação do endereço destino para o envio dos dados, a primitiva a ser utilizada será *sendto(descritor, mensagem, tamanho, controle, destino, tdest).* Os quatro primeiros argumentos são os mesmos da primitiva *send*. Os dois últimos especificam o endereço destino usando a mesma estrutura *endloc* da primitiva *bind*.

Devido ao grande número de argumentos da primitiva *sendto*, o que aumenta sua complexidade na utilização, foi definida a primitiva *sendmsg(descritor, estmsg, controle)*. O argumento *estmsg* é uma estrutura de dados que contém informações da mensagem a ser enviada, seu comprimento, o endereço destino e o comprimento do endereço destino.

3.3.8 Recepção de Dados Através de um Socket

O recebimento de dados através de um *socket* poderá ocorrer através de cinco primitivas básicas (*read, readv, recv, recvfrom e recvmsg*).

Abaixo estão demonstradas as sintaxes de cada primitiva. A definição de cada argumento não será necessária, uma vez que suas funções são similares às que são definidas nas primitivas de envio de dados, tornando sua compreensão possível através desta abstração:

1) *read(descritor, buffer, tamanho)*;
2) *readv(descritor, iovector, tamvetor*);
3) *recv(descritor, buffer, tamanho, controle*);
4) *recvfrom(descritor, buffer, tamanho, controle, origem, torigem)*;
5) *recvmsg(descritor, estmsg, controle)*.

3.3.9 Obtenção de Endereços Locais e Remotos de Sockets

Para que uma aplicação possa determinar o endereço local e destino em uma conexão a partir de *sockets* já criados, ela deverá utilizar as primitivas *getpeername* e *getsockname*. A primitiva *getpeername* fornece informações do endereço remoto, no qual o *socket* está conectado. Sua utilização será da seguinte forma: *getpeername(descritor, destino, tdest)*. O descritor especifica o *socket*

para o qual o endereço é desejado. O argumento destino segue as especificações da estrutura de endereçamento já comentada e *tdest* é um ponteiro para um inteiro que vai conter o comprimento do endereço.

A primitiva *getsockname(descritor, endloc, tend)* é utilizada quando se deseja obter informações do endereço local ao qual o *socket* está conectado.

3.3.10 Outros Procedimentos Relacionados a Sockets

As primitivas de uma API contém vários procedimentos. Por exemplo, após um servidor chamar o procedimento *accept* para aceitar uma requisição recebida, o servidor pode chamar o procedimento *getpeername* para obter o endereço completo do cliente remoto que iniciou a conexão. Um cliente também poderá chamar o procedimento *gethostname* para obter informações sobre o computador em que ele está executando. Existem ainda duas primitivas de uso geral que podem configurar opções de um *socket* (*setsockopt*), ou obter informações sobre o mesmo (*getsockopt*).

Os procedimentos *gethostbyname* e *gethostbyaddr* são utilizados quando um usuário necessita obter o endereço IP a partir do nome do computador, ou então obter o nome do computador a partir do endereço IP.

3.3.11 Modelagem da Conexão Via Socket entre Aplicações Cliente/Servidor

Na FIGURA 16 a seguir estão especificadas as principais primitivas que fazem parte da implementação de uma aplicação com comunicação baseada em *sockets*.

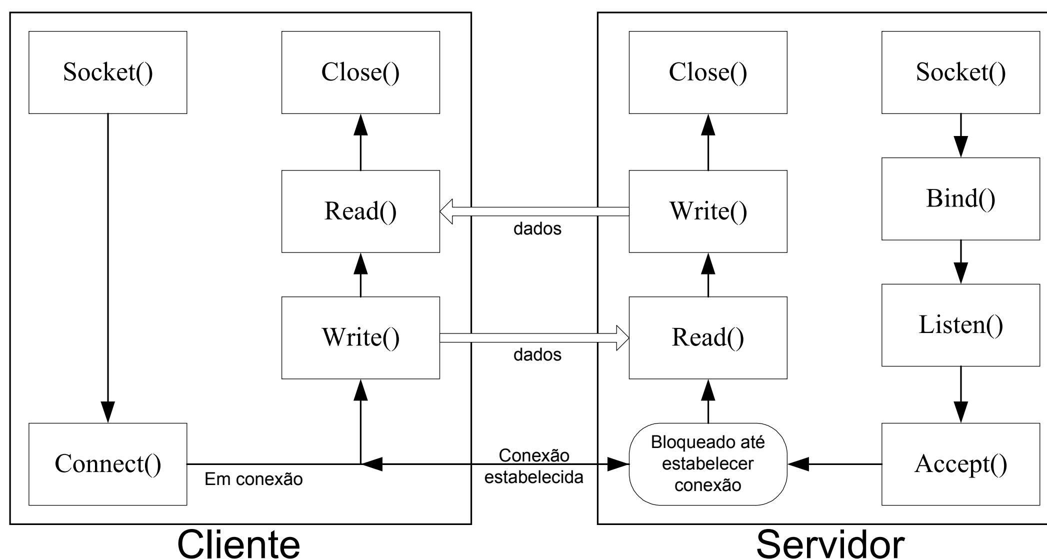


FIGURA 12 – Primitivas de Comunicação Cliente/Servidor Via Sockets

A FIGURA 12 demonstra um esquema básico de utilização das primitivas de *socket* para a construção de um aplicativo servidor ou e de um aplicativo cliente. O uso de cada primitiva está detalhado logo a seguir:

1) Tanto o aplicativo cliente quanto o aplicativo servidor criam os pontos finais de comunicação com a primitiva *socket (endereço ip, porta)*; ressalta-se que o endereço *ip* identifica uma única máquina e a porta para uma aplicação em uma determinada máquina;
2) O servidor associa uma porta ao *socket* criado, através da primitiva *bind()* e habilita um *buffer* à espera de novos pedidos de conexões;
3) O servidor bloqueia a estação (primitiva *accept ()*) até que um pedido de conexão chegue (primitiva *connect()*);
4) O Cliente faz uma requisição para estabelecer conexão (primitiva *connect()*);
5) O Servidor aceita a conexão;
6) Início da transferência de dados entre as duas entidades (primitivas *read()* e *write()*);
7) O Cliente e servidor fecham o *socket*.

O uso de *sockets* com um protocolo não orientado à conexão é semelhante a uma conexão orientada, exceto pelo fato de que não é necessário estabelecer um canal para a transferência de dados, isto é, as primitivas *accept ()* e *connect ()* não são utilizadas.

3.4 Modelagem da API Network

A modelagem da API *Network* foi desenvolvida usando a ferramenta Rational Rose [16], que utiliza o método de modelagem UML (*Unified Modeling Language*) [21]. Esta ferramenta oferece diversos níveis e estruturas para a modelagem. Entretanto, devido à natureza deste projeto, que é dirigido à criação de classes para utilização como uma API, o principal enfoque utilizado da ferramenta foi do Diagrama de Classes.

Na FIGURA 13 é apresentada a modelagem estrutural dos pacotes da API *Network*. Nesta figura pode-se observar os dois pacotes que são detalhados nas seções seguintes.

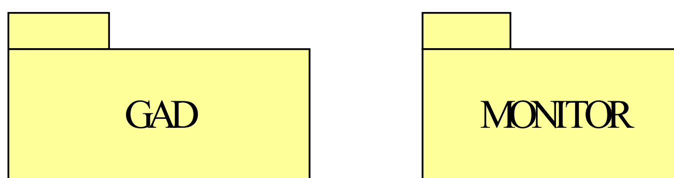


FIGURA 13 – Modelagem Básica da API *Network*

3.4.1 Pacote GAD

O primeiro dos pacotes apresentados na FIGURA 13 é o GAD (Gerador de Aplicações Distribuídas). Este pacote contém várias classes que irão fornecer as funcionalidades necessárias para a implementação de aplicações cliente/servidor através de *sockets*. Estas classes são descritas nas seções a seguir.

3.4.1.1 Classe GetXbY

Nessa classe estão os métodos que serão responsáveis pelo fornecimento de informações referentes a um computador remoto, através de seu nome ou de seu endereço IP, de um determinado protocolo da família TCP/IP e de um determinado serviço na camada de aplicação. Desses métodos podemos citar:

- *GhostbNameProc()* – Fornece o endereço IP do computador a partir do nome conhecido do mesmo;
- *GhostbNumProc()* – Fornece o nome do computador a partir do endereço IP conhecido;
- *GprotbName()* – Fornece informações do protocolo a partir de seu nome conhecido;
- *GprotbNum()* – Fornece informações do protocolo a partir do seu código numérico;
- *GservbName()* – Fornece informações do nome do serviço utilizado pelo protocolo em questão;
- *GservbPort()* – Fornece informações do serviço utilizado pelo protocolo em uma porta específica.

3.4.1.2 Classe Finish

Nessa classe foram implementados os métodos que serão responsáveis pelo fechamento do *socket* utilizado e encerramento da conexão que tiver sido estabelecida entre os *sockets* através das portas de origem e destino. Os principais métodos dessa classe são:

- *CloseSockProc()* – Fecha o *socket* que foi aberto na aplicação;
- *ShutdownProc()* – Encerra a conexão que foi estabelecida através das portas nos *sockets*.

3.4.1.3 Classe Function

Todos os métodos públicos que são utilizados por outras classes foram implementados na Classe *Function*, buscando com isto facilitar a modelagem do pacote. Os métodos mais importantes desta classe estão relacionados a seguir:

- *LoadStartupOption()* – Inicializa as opções que foram definidas como padrão na geração de aplicações cliente/servidor;

- *LoadDefaultWinsock()* – Carrega a versão padrão da *DLL Winsock* definida na implementação da aplicação;
- *CheckLib()* – Verifica se a *DLL Winsock* está carregada antes da execução de suas primitivas;
- *LoadLib()* – Carrega a versão da *DLL Winsock* que está sendo requisitada;
- *CheckWinsock()* – Identifica se a versão da *DLL Winsock* é compatível com a função que estiver sendo requisitada;
- *StringtoAddress()* – Converte uma cadeia de caracteres em um formato de endereço IP exigido pelo procotolo;
- *GetSocket()* – Procura por um *socket* na lista de *sockets* abertos;
- U*pdateSocket()* – Atualiza a lista de *sockets* abertos;
- *GetError()* – Fornece o código do erro gerado por um método;
- *WSAError()* – Retorna a descrição do erro fornecido pelo método *GetError*, conforme definições da *DLL Winsock*;
- *NotImplemented()* – Validação dos métodos implementados a partir das primitivas da *DLL Winsock*.

3.4.1.4 Classe Open

Esta é a classe responsável pelas operações de inicialização e estabelecimento de conexões entre aplicações cliente e servidor. Esses métodos poderão ser utilizados separadamente ou então através da invocação de operações responsáveis pela otimização dos mesmos. Abaixo estão descritos os métodos instanciados por esta classe:

- *WStartupProc()* – Inicializa a *DLL Winsock* carregada anteriormente;
- *SocketProc()* – Responsável pela definição e criação de um *socket*;
- *ListenProc()* – Define a lista onde vão ser armazenadas as requisições de computadores remotos;
- *BindProc()* – Associa o *socket* a uma porta de origem, que computadores remotos irão utilizar para a transmissão de dados;
- *AcceptProc()* – Método utilizado por aplicações do tipo servidor. Prepara o servidor para receber conexões de cliente;

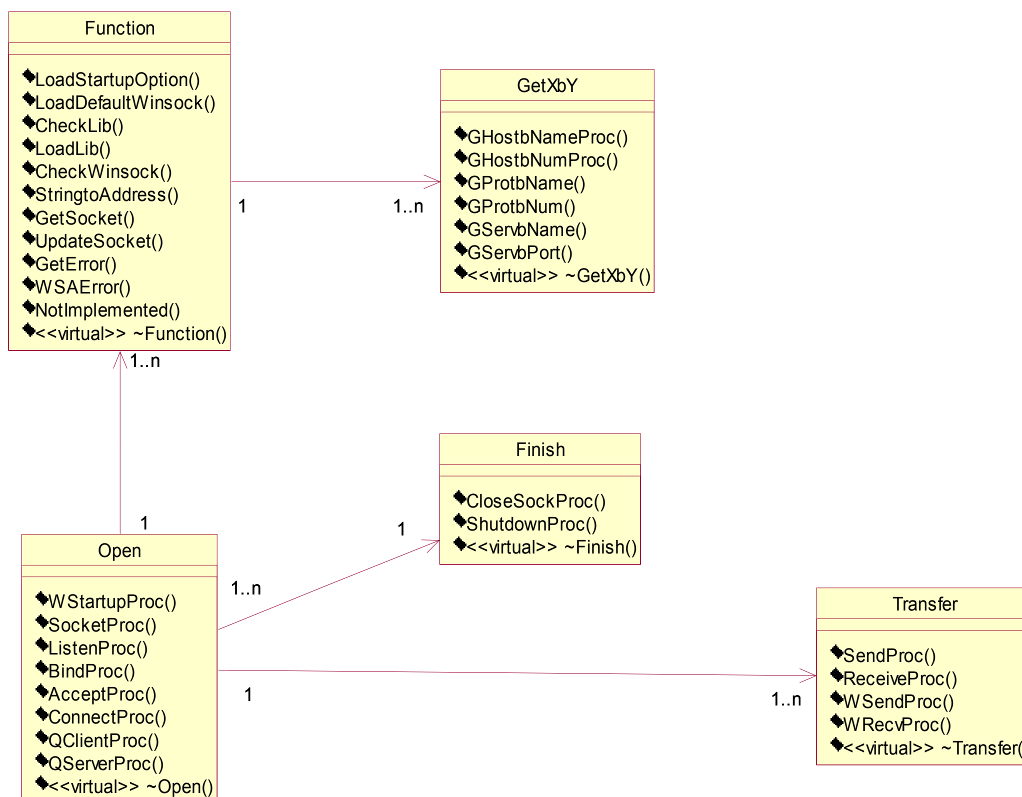
- *ConnectProc()* – Método utilizado por aplicações do tipo cliente. Responsável pela solicitação de conexão a um computador que está executando a aplicação servidor;
- *QClientProc()* – Procedimento que agrega os métodos necessários para a criação de uma aplicação do tipo cliente;
- *QServerProc()* – Procedimento que agrega os métodos necessários para a criação de uma aplicação do tipo servidor.

3.4.1.5 Classe Transfer

Nesta classe foram implementados os métodos responsáveis pelo processo de transferência de dados entre uma aplicação servidor e uma aplicação cliente. Esses métodos estão divididos em duas sub-classes, sendo que a primeira contém os métodos responsáveis pelos processos de transferência de dados que utilizam protocolos orientados a conexão; a segunda contém os métodos para serem utilizados por processos que utilizam protocolos não orientados a conexão. Os métodos desta classe estão relacionados a seguir:

- *SendProc()* – Método responsável pelo envio de pacotes em conexões orientadas ;
- *ReceiveProc()* – Método responsável pela recepção de pacotes em conexões orientadas.
- *WSendProc()* - Método responsável pelo envio de datagramas em conexões não orientadas ;
- *WRecvProc()* – Método responsável pela recepção de datagramas em conexões não orientadas.

3.4.1.6 Diagrama de Classes do Pacote GAD



3.4.2 Pacote MONITOR

O segundo dos pacotes apresentados na FIGURA 13 é o MONITOR. Este pacote contém várias classes que irão fornecer as funcionalidades necessárias para a monitoração do tráfego existente entre as aplicações cliente e servidor através de sockets. Estas classes são descritas nas seções a seguir.

3.4.2.1 Classe CDigDispositivo

Para que o Pacote Monitor possa atuar no dispositivo de rede instalado no equipamento, foram adicionados nessa Classe métodos que têm como funcionalidades reconhecer e configurar a interface de rede pela qual os pacotes serão capturados. Entre esses métodos podemos citar:

- *ObtemEndereco()* – Verifica qual é o dispositivo de rede instalado e obtém o endereço IP do mesmo;
- *ConfiguraEndereco()* – Configura o dispositivo para atuar no modo promíscuo e assim possibilitar a captura de todos os pacotes que trafegarem em sua rede.

3.4.2.2 Classe CDlgFiltro

Nesta classe estão os métodos responsáveis por configurar o dispositivo de rede para capturar pacotes segundo critérios pré-estabelecidos, tais como por palavra, por endereço de origem ou destino, por protocolo, entre outros. Desses métodos podemos descrever:

- *FiltraIP()* – Configura o filtro para a captura de pacotes que possuam em seus campos de origem e destino os IP selecionados;
- *FiltraProtocoloAplicacao()* – Configura o filtro para a captura de pacotes dos procolos de aplicação especificados;
- *FiltraProtocoloTransporte()* – Configura o filtro para a captura de pacotes dos protocolos de transporte especificados;
- *FiltraPalavra()* – Configura o filtro para a captura de pacotes que possuam em seu campo de dados as palavras que foram especificadas.

3.4.2.3 Classe CCapturaPacote

Classe responsável pelo processo de captura dos pacotes a partir do dispositivo selecionado, com ou sem um filtro, especificando os critérios de seleção. Os métodos desta classe são:

- *ConfiguraInterface()* – Configura o dispositivo de rede segundo os critérios estabelecidos pelas classes CdlgDispositivo e CdlgFiltro;
- *IePacote()* – Médoto para a captura dos pacotes trafegados no dispositivo de rede e armazenamento dos mesmos em um *array* de memória.

3.4.2.4 Classe CEstatistica

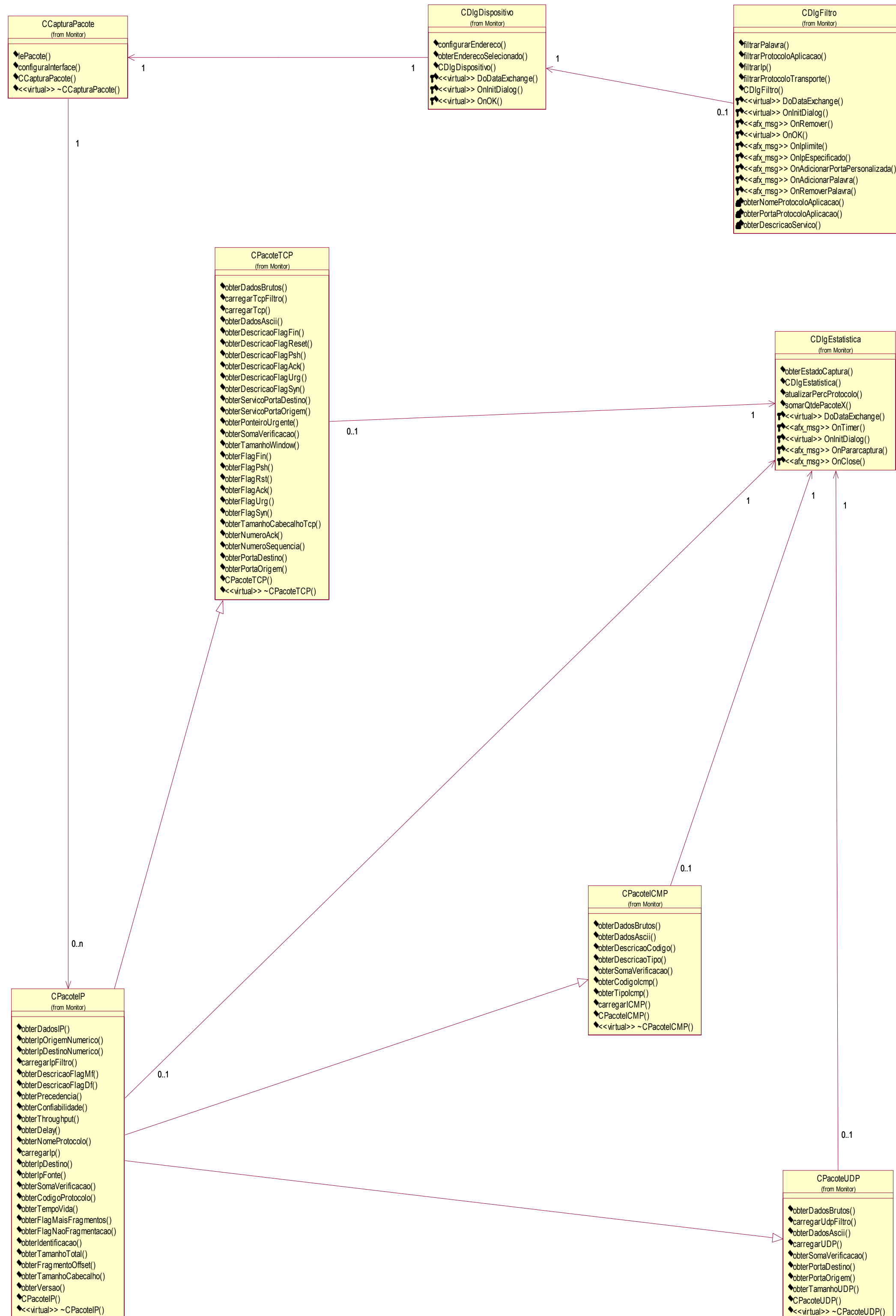
Esta classe possui os métodos necessários para a visualização estatística das quantidades e porcentagens dos pacotes capturados, separados por tipo de protocolo. Entre estes métodos podemos citar:

- *ObterEstadoCaptura()* – Obtém, a partir do dispositivo de rede, o estado em que se encontra a captura, isto é, em andamento, aguradando e interrompido;
- *SomarQtdePacote()* – Totaliza os pacotes capturados por tipo de protocolo;
- *AtualizaPercProtocolo()* – Totaliza os pacotes nas devidas porcentagens pelo tipo de protocolo em relação à quantidade total capturada.

3.4.2.5 Classes CPacoteIp, CPacoteTCP, CPacoteUDP e CPacoteICMP

Estas classes são responsáveis pela decodificação do pacote capturado. No caso da classe CPacoteIP, os métodos que foram implementados na mesma têm como finalidade a obtenção de cada campo pertecente a ele, bem como a separação de cabeçalhos que especificam outros tipos de pacotes, como é o caso dos pacotes TCP, UDP e ICMP. Estes, por sua vez, foram tratados como generalizações nas classes que os decodificam, visando com isso, facilitar a modelagem de um outro tipo de protocolo que venha a ser incorporado no Monitor. Esta generalização, pode ser constatada no Diagrama de Classes do item a seguir.

3.4.2.6 Diagrama de Classes do Pacote MONITOR



3.5 Resumo do Capítulo

Neste capítulo foram abordados os conceitos que envolvem a comunicação entre processos de aplicações baseadas no paradigma Cliente/Servidor. Entre os mecanismos de comunicação disponíveis, como RPC, Corba ou Com/DCom, foi selecionado o que utiliza a transferência via *Sockets*, uma abstração comum nos sistemas operacionais mais utilizados atualmente, como o Windows e Linux. Este mecanismo foi escolhido por suas facilidades envolvendo aplicações distribuídas em uma rede, como velocidade de transferência de dados e simplicidade na implementação.

A partir do mecanismo selecionado, foi modelada e implementada a API *Network*, composta por dois Pacotes de Classes, o GAD (Gerador de Aplicações Distribuídas) e o Monitor (Monitoração de Tráfego), contendo os métodos necessários para o desenvolvimento de aplicações cliente/servidor e também para a monitoração dos pacotes que forem gerados e transferidos pelas mesmas. Como ambiente de modelagem foi utilizada a ferramenta Rational Rose, que disponibiliza a metodologia de análise orientada a objetos denominada UML.

No próximo capítulo, será descrito a implementação de um Ambiente utilizando a API *Network* na geração e monitoração de aplicações distribuídas em uma rede de computadores.

Capítulo 4

AGAD – AMBIENTE PARA GERAÇÃO DE APLICAÇÕES DISTRIBUÍDAS

4.1 AGAD - Ambiente para Geração de Aplicações Distribuídas

A seguir, serão detalhados os passos utilizados para a implementação do AGAD, que foi modelado de uma forma que o usuário possa optar pela construção de aplicações denominadas Servidor, responsáveis por suprir aplicações Cliente de serviços específicos, respondendo às requisições com o envio das informações solicitadas. Ele é composto de três módulos principais, que interagem com o sistema operacional através da DLL *Winsock*, conforme demonstrado na FIGURA 14, a seguir:

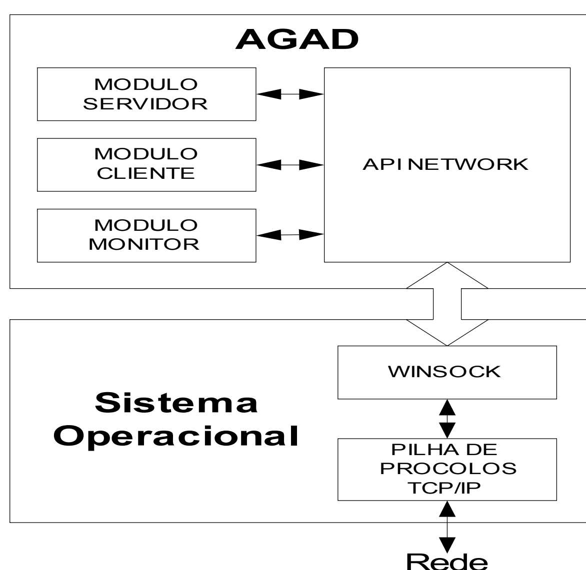


FIGURA 14 – O AGAD e sua Conexão com o Sistema Operacional

O Módulo Servidor contém as funções necessárias à construção de aplicações Servidor, disponibilizadas individualmente ou em modelos predefinidos. A construção da aplicação poderá ocorrer passo a passo, onde o usuário irá controlar a utilização das funções e suas propriedades, ou então, utilizando os modelos predefinidos contendo conjuntos de funções necessárias

para a prestação do serviço requerido. A execução das funções, ou do modelo escolhido, irá ocorrer imediatamente após sua requisição;

O Módulo Cliente foi baseado nos mesmos conceitos utilizados na elaboração do Módulo Servidor para possibilitar aos usuários as mesmas facilidades obtidas na geração de aplicações Cliente. Nestes dois módulos, ao final da formulação das aplicações, será gerado um Fonte, contendo as linhas de código em linguagem C, para ser compilado separadamente do Ambiente.

O Módulo Monitor foi elaborado com funcionalidades de monitoria e depuração do processo de comunicação das aplicações geradas nos módulos anteriores. Através dele são capturados todos os pacotes gerados na transmissão e recepção, disponibilizando-os em memória para a visualização e análise do comportamento das aplicações.

4.2 Descrevendo o AGAD

Como já mencionado, este Ambiente foi desenvolvido para dar suporte à API *Network*, criando um ambiente visual para ser utilizado na construção de aplicações distribuídas com comunicação via *Socket*. O ambiente disponibiliza as funções da API em forma de requisições ou opções em menus, separados em grupos de acordo com suas funcionalidades. Estes grupos podem ser vistos na FIGURA 15, a seguir.

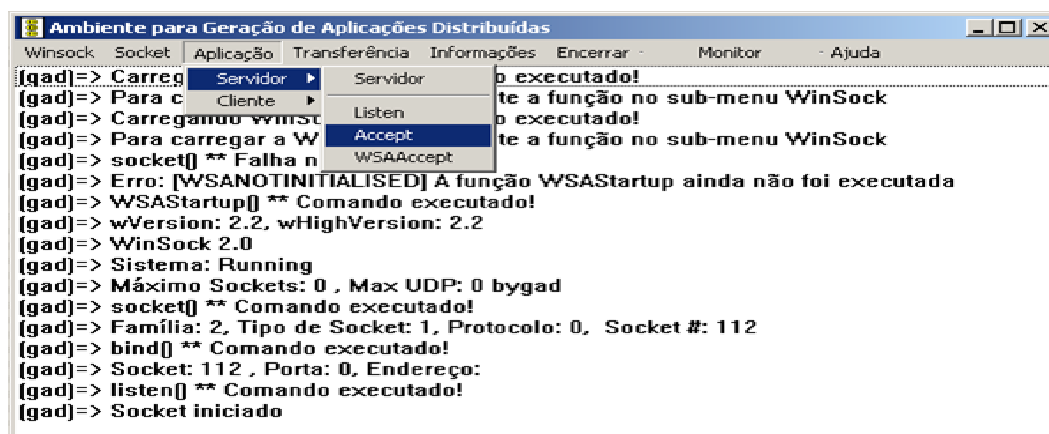


FIGURA 15 – Ambiente para Geração de Aplicações Distribuídas

Este ambiente oferece ao usuário as principais funções para a criação de aplicações cliente ou servidor, que utilizam a abstração *Socket* para a

comunicação com os protocolos da rede. Em sua utilização, as funções conforme forem sendo selecionadas, executarão suas operações de forma seqüencial, ou seja, na ordem em que forem chamadas a partir dos menus. Caso uma determinada função seja pré-requisito para a que está sendo selecionada, o Ambiente irá comunicar o usuário através de mensagens específicas, inclusive em caso de erros. No final da composição da aplicação, após o usuário constatar as execuções das funções selecionadas na ordem correta e com todos os seus parâmetros validados, será gerado um arquivo texto contento a aplicação em linguagem C, que poderá ser modificada segundo necessidades mais específicas e compilado separadamente, ou então ser adaptado e utilizado em sistemas mais complexos.

Nesta fase de implementação, cada execução do Ambiente irá ser responsável pela composição de uma aplicação, seja ela Cliente ou Servidor, conforme mencionado.

4.2.1 Monitorando o Tráfego entre Aplicações Distribuídas

Uma ferramenta eficiente de monitoração deve ser capaz de reconhecer qualquer tipo de chamada às primitivas residentes nas aplicações cliente/servidor que utilizam a interface *socket*, identificar seus argumentos e seus valores durante a execução, bem como relacionar através da aplicação os processos em execução, a interação entre eles e quais protocolos estão em atividade no momento da verificação. Sendo assim, foi construída uma aplicação denominada Módulo de Captura de Tráfego, na qual, após a execução das aplicações Cliente/Servidor geradas pelo AGAD, serão capturados todos os pacotes trocados entre elas. Com isto, o usuário poderá obter informações valiosas com relação às conexões envolvidas, aos resultados da utilização das funções de transmissão e recepção e com elas poder observar o fluxo ocasionado nos protocolos selecionados, bem como gerar relatórios estatísticos baseados nos campos dos pacotes capturados e armazenados em um buffer pelo módulo. Este módulo deverá ser acessado através da opção “MONITOR” do menu do AGAD, conforme demonstrado na FIGURA 17.

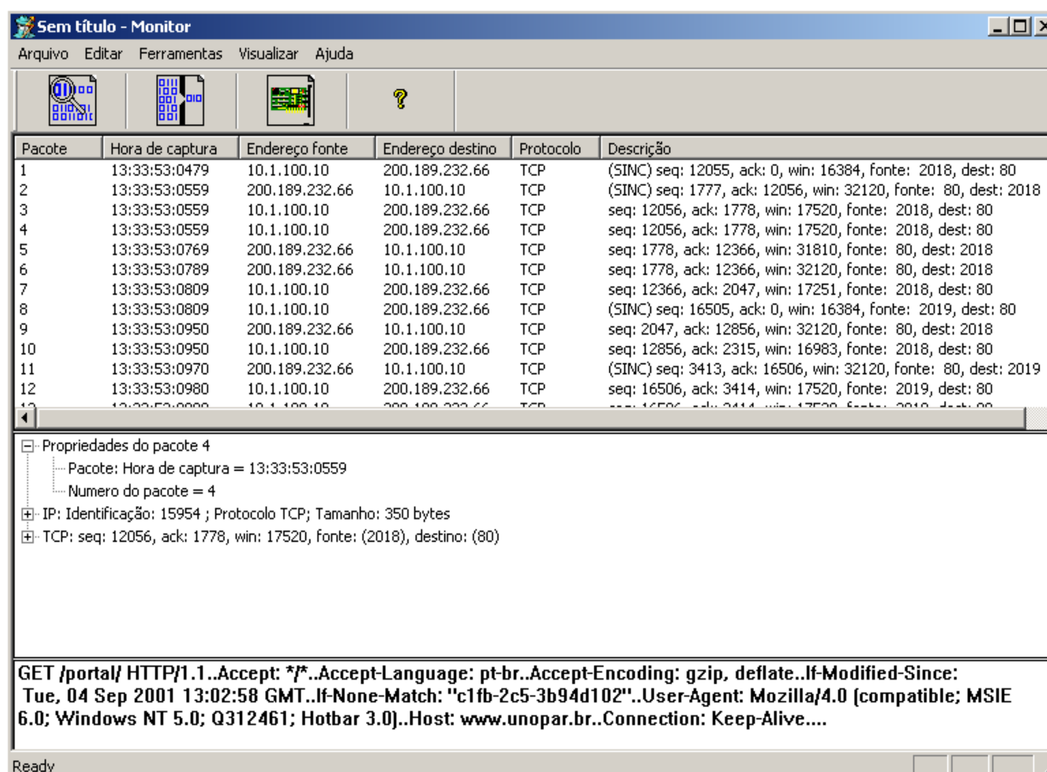


FIGURA 16 - Módulo Monitor

O protótipo da ferramenta acima foi modelado segundo as informações que estão disponibilizadas no datagrama IP, capturado de uma rede *Ethernet*. Para que o usuário possa selecionar quais tipos de pacotes deverão ser capturados, foi desenvolvido na ferramenta um sistema de filtros, nos quais poderão ser escolhidos os critérios de seu interesse, como por exemplo, os endereços IP envolvidos, o protocolo, portas, palavras chaves dentro do campo de dados dos pacotes, entre outros, conforme demonstrado na FIGURA 18:

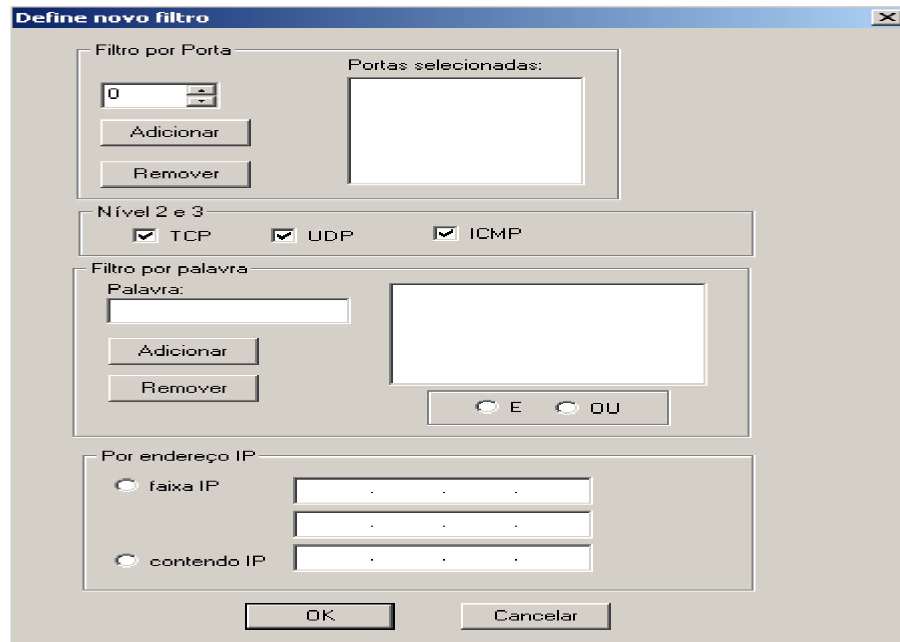


FIGURA 17 – Função do Monitor para Definição de Critérios para Captura de Pacotes

4.3 Resumo do Capítulo

Neste capítulo foram descritas as etapas da modelagem e implementação do AGAD (Ambiente para Geração de Aplicações Distribuídas), a partir da API *Network* definida no Capítulo 3. Foram abordados os aspectos relevantes do AGAD, como sua construção e sua utilização na geração de aplicações baseadas no paradigma cliente/servidor, utilizando o mecanismo definido como *Sockets*, já descrito anteriormente. No próximo capítulo, será proposto um modelo de integração deste ambiente com o TEV (*Teaching Environment for Virtuoso*), visando a interconexão das aplicações desenvolvidas nos mesmos.

Capítulo 5
INTEGRAÇÃO DO AGAD COM O AMBIENTE TEV

5.1 Descrevendo a Interação do AGAD com o Ambiente Visual TEV

Este capítulo propõe a modelagem de um mecanismo que viabiliza a integração do AGAD com o TEV (*Teaching Environment for Virtuoso*), incorporando ao mesmo novas funcionalidades que possibilitem a implementação de sistemas de tempo e sua interconexão com as aplicações desenvolvidas no AGAD, visando prover a este ambiente novos recursos para a sua utilização na geração de sistemas distribuídos, protocolos de comunicação, aplicações cliente/servidor e monitoração das primitivas de comunicação. Estes ambientes, além de sua utilização por projetistas no desenvolvimento de sistemas, poderão também ser utilizados como auxiliares no ensino dos conceitos nos quais eles foram fundamentados, ou seja, aplicações de tempo real e sistemas distribuídos. Sendo assim, será este o enfoque principal adotado para o uso destes ambientes neste trabalho. A seguir estão descritos alguns tópicos que foram abordados na implementação desta integração.

5.2 TEV – Teaching Environment For Virtuoso

O objetivo do Ambiente Visual é a elaboração de um conjunto de ferramentas integradas que auxiliam no desenvolvimento de programas paralelos de tempo real para serem executados em uma plataforma com o Virtuoso[1], que é um *kernel* desenvolvido pela *Eonic Systems, Inc.* para a criação de aplicações de tempo-real, que precisam manipular eventos externos baseadas em restrições de tempo. Este *kernel* tem como objetivo oferecer às aplicações de tempo-real, a capacidade de utilizar várias tarefas e prover a comunicação entre as mesmas. Além da habilidade de manipulação de interrupções com a mínima perda de tempo, fazendo com que a aplicação de tempo-real possa responder de maneira desejada a eventos externos.

---

[1] Virtuoso is now a trademark of Wind River System Inc.

O Virtuoso oferece um sistema de programação organizada em diversos níveis, como mostra a FIGURA 19.

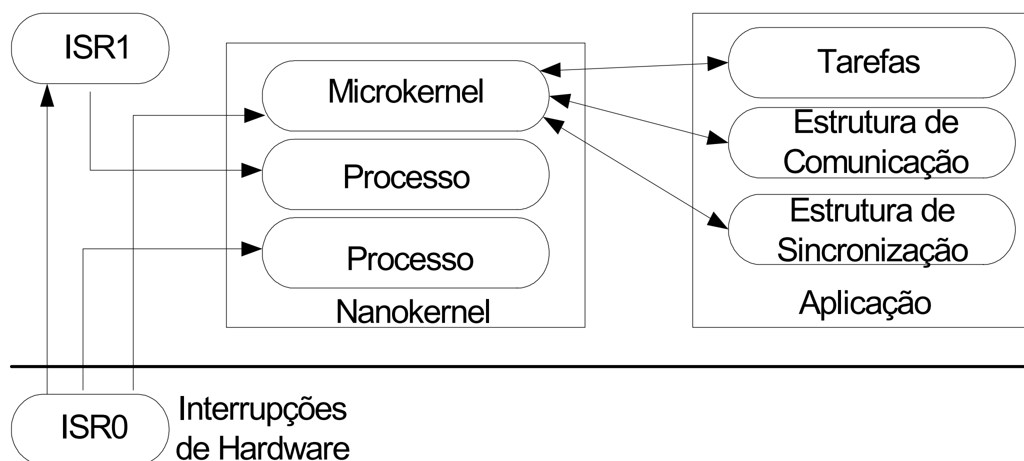


FIGURA 18 – Níveis de Programação do *Ker*nel Virtuoso

Os níveis ISR (*Interrupt Service Routine*) são usados para controlar as interrupções de hardware do processador.

O nível *Nanokernel* é composto por diversas tarefas de contexto reduzido chamadas processos. Estes processos são rotinas em linguagem *assembly,* que podem chamar funções em linguagem C. O processo que gerencia o escalonamento de tarefas é chamado de *microkernel.*

O *microkernel* é constituído por mais de setenta serviços que podem ser chamados a partir da linguagem C. Os componentes básicos deste nível de programação são os objetos do *microkernel*, que são instâncias de classes predefinidas pelo Virtuoso, constituídas de um conjunto de atributos e operações .

Durante a fase de desenvolvimento, o programador define o conjunto de objetos do *microkernel* que sua aplicação irá utilizar, alocando-os em processos específicos. Em um sistema monoprocessador, estes objetos estão localizados em um mesmo nó. Em um sistema paralelo, estes objetos poderão estar distribuídos entre os diversos processadores do sistema.

Os principais objetos do *microkernel* são as tarefas, os semáforos, os *mailboxes*, as filas, os mapas de memória, os recursos e os *timers.*

Este *kernel*, apesar de oferecer os recursos encontrados em uma linguagem de programação paralela, não oferece um ambiente de alto nível que facilite a utilização destes recursos no desenvolvimento das aplicações. Para sanar esta deficiência, foi desenvolvido o TEV (*Teaching Enviroment for Virtuoso*) [10,11] em um trabalho de mestrado anterior, que oferece ao usuário uma interface amigável e auto-explicativa para o desenvolvimento de aplicações de tempo real, executadas na máquina paralela com o auxílio do *kernel* Virtuoso. O TEV por sua vez, faz parte de um Ambiente Integrado de Desenvolvimento, que pode ser visualizado na FIGURA 20.

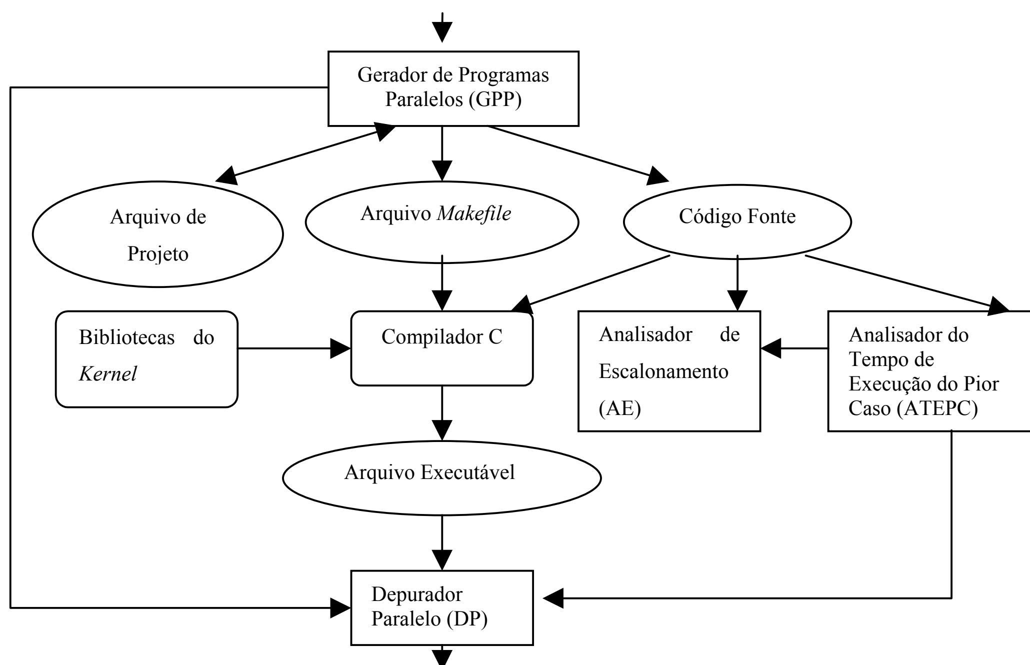


FIGURA 19 – Componentes do TEV

Os retângulos denotam suas ferramentas. Os arquivos gerados em cada estágio são representados por elipses. Os retângulos com cantos arredondados representam as ferramentas extras com as quais o Ambiente Visual interage.

5.2.1 Gerador de Programas Paralelos (GPP)

Atendendo aos objetivos deste projeto, foi estudado com mais detalhe o GPP, por ser este o sistema que será integrado ao Ambiente AGAD. A

função desta ferramenta é auxiliar na geração de código fonte dos programas executados na máquina paralela. As aplicações são geradas em um modelo gráfico utilizado também pelas outras ferramentas do Ambiente Visual. Este modelo é representado por um grafo, onde os nós denotam as estruturas de dados do programa paralelo (tarefas, semáforos, *mailboxes*, recursos, etc.) e as arestas representam as operações de comunicação e sincronização relacionadas com estas estruturas. O modelo pode ser complementado pelo usuário, através da inclusão de descrições textuais na linguagem de programação C. O modelo gráfico mais as descrições textuais são armazenadas em arquivos do tipo *makefile*, utilizados no processo de compilação e *linkedição* do código fonte produzido.

A primeira versão do GPP foi chamada de TEV – *Teaching Environment for Virtuoso*, conforme mostrado na FIGURA 21.

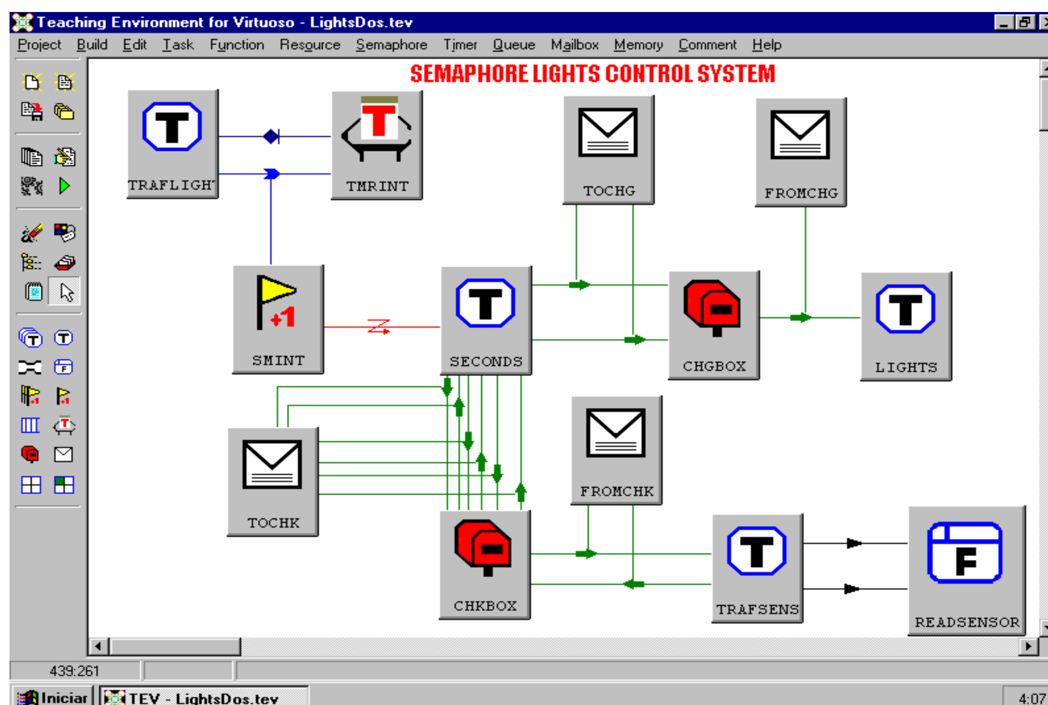


FIGURA 20 – Teaching Environment for Virtuoso

Nos próximos tópicos, serão estudados alguns conceitos que foram necessários para efetuar a interação entre o TEV e o AGAD, possibilitando assim o uso dessas ferramentas de forma integrada, permitindo que aplicações geradas pelas mesmas possam ser executadas com a possibilidade de transferência de dados entre elas.

5.3 Mecanismos Utilizados no Compartilhamento de Dados entre Aplicações Distribuídos

A API Win32 da Microsoft oferece mecanismos para facilitar compartilhamento de dados e comunicação entre processos. Tipicamente aplicativos podem utilizar o IPC (*Interprocess Communications*) sendo categorizados como clientes ou servidores, seguindo o modelo cliente-servidor como descrito anteriormente. Entre esses mecanismos será utilizado o *Pipe* [7], por ser este o modelo utilizado pelo *Kernel* Virtuoso e suas aplicações.

Existem dois tipos de *pipes*, os anônimos e os com nome. Para os propósitos desse projeto, será trabalhado apenas com os *pipes* com nome, que são utilizados para transferir dados entre processos que não estão relacionados e para transferir dados entre processos em computadores diferentes. O servidor cria o *pipe* com um nome conhecido, e os clientes se conectam a ele, e podem passar a se comunicar com o servidor através de operações de escrita e leitura.

5.3.1 A Interação entre Processos Através dos Canais do Virtuoso

Um canal do Virtuoso pode ser interpretado como uma fila do tipo FIFO (*First In, First Out*) unidirecional, e desta forma ser utilizado para passar informações entre tarefas.

Na versão 4.2 do Virtuoso existem dois tipos de canais: canais alvo e canais hospedeiros de entrada e saída. Canais alvo são objetos estáticos, criados no gerente de projetos do Virtuoso, enquanto canais hospedeiros são criados dinamicamente. Para esse projeto, foram utilizados apenas os canais hospedeiros, que não trabalham com *buffers* e permitem que as mensagens sejam transferidas diretamente entre as tarefas envolvidas.

Levando-se em consideração que a utilização de canais permite uma comunicação rápida e assíncrona entre as tarefas, alem de poder acessar outros tipos de recursos do hospedeiro, tal como *pipes* e *sockets*, foram propostos dois modelos baseados neste mecanismo. O primeiro a partir da alteração de algumas funções existentes no *Kernel* Virtuoso e conseqüentemente no TEV,

possibilitando com isto o desenvolvimento de aplicações distribuídas segundo o paradigma cliente/servidor diretamente no TEV. O segundo foi modelado a partir da implementação de uma biblioteca de funções denominada API *Network,* a qual foi integrada a uma aplicação denominada AGAD (Ambiente para Geração de Aplicações Distribuídas) utilizando o IPC *Sockets* como mecanismo de comunicação entre as tarefas, em seguida esta ferramenta foi integrada ao TEV para que as aplicações desenvolvidas a partir destas duas ferramentas pudessem se comunicar através dos canais do *kernel*.

A seguir, serão detalhados os dois modelos que foram estudados, com a adoção do segundo por ser o que melhor que adequou as necessidades deste trabalho.

5.4 Implementação dos Mecanismos de Interação entre o AGAD e o TEV

Conforme descrito no capítulo 4, o AGAD foi implementado utilizando o mecanismo IPC *socket* para a comunicação entre as tarefas das aplicações geradas no mesmo, localmente ou em máquinas distintas. Já as tarefas das aplicações geradas pelo TEV se comunicam basicamente através de canais ou filas. A seguir será demonstrado o processo responsável pela comunicação de uma tarefa gerada no AGAD com uma tarefa gerada no TEV, de forma a não alterar as características fundamentais dessas ferramentas, para com isso poderem ser utilizadas em conjunto ou em separado.

Como visto anteriormente, não foi possível a implementação de servidores e clientes *sockets* diretamente no *kernel* a partir do TEV, portanto, chegou se a conclusão que eles deverão ser implementados como processos separados, que se comunicam com o TEV através de um mecanismo do tipo IPC, nesse caso, utilizando *pipes* com nomes conhecidos. Para isso, foi necessário alterar o código do AGAD, adicionando à ferramenta a capacidade de ler o código fonte gerado pelo TEV e dele extrair quais são os canais de entrada e saída especificados no mesmo. No entanto, a extração será possível mediante a definição dois tipos de dados: *chaninnm_t* (nome do canal de entrada) e *chanoutnm_t* (nome do canal de saída), que são ponteiros para cadeias de

caracteres. Com a utilização dessas estruturas na especificação dos nomes dos pipes, o AGAD através do algoritmo de leitura e seleção implementado e demonstrado na FIGURA 22, identificará com facilidade os canais que poderão ser utilizados por suas aplicações na interação com as aplicações geradas no TEV.

```
while( !feof( fp ) ) {
    fgets( buf, 80, fp );
    if ( strstr( buf, "chaninnm_t" ) ) {
       if (strstr( buf,"typedef" ) == NULL ) {
          strtok( buf,"\"" );
          temp1=strtok( NULL,"\"" );
          temp2=strtok( NULL,"\"" );
          size= temp2 - temp1;
          strncpy( nameinlist[ i++ ], temp1, size );
}}}
```

FIGURA 21 - Identificação dos Canais no Código Fonte do TEV

Na porção de código apresentada na FIGURA 22 não está listado a abertura do arquivo, que em geral é o *tasks.c* contido no diretório do projeto do TEV. Uma vez o arquivo aberto, suas linhas serão lidas seqüencialmente, verificando se as cadeias *chaninnm_t* e *chanoutnm_t* estão contidas nas mesmas, para então fazer uso da função *stdtok()* da biblioteca padrão da linguagem C, visando selecionar somente os nomes dos canais que deverão em seguida, serem exibidos aos usuários do AGAD durante a geração de aplicações distribuídas baseadas no paradigma cliente/servidor. No Virtuoso, os canais são abertos apenas com definição na função criadora do nome pelo qual o mesmo será conhecido, já no Windows 2000 e NT as aberturas dos canais devem seguir o formato "//computador/pipe/nome_do_canal" ou simplesmente "//./pipe/nome_do_canal" quando estiverem sendo referenciados na máquina local.

Sendo então conhecidos os canais que foram abertos pela aplicação desenvolvida no TEV, o usuário da ferramenta AGAD poderá selecionar o canal de entrada e o canal de saída para serem inicializados através de uma função chamada *CreateNamedPipe()*, para em seguida serem conectados entre si através

da função *ConnectNamedPipe()*, bloqueando-os até que a conexão seja estabelecida.

Os passos acima foram necessários para o estabelecimento da conexão entre tarefas das aplicações geradas pelo AGAD e pelo TEV. A seguir estão demonstradas as funcionalidades que proporcionaram a capacidade de transferência de dados entre essas tarefas. Isso foi possível através da alteração das funções de *send* e *receive* contidas no AGAD, de forma a não alterar a premissa básica que é a comunicação através de *sockets*. Foram adicionados códigos a essas funções, mais especificamente nas funções *send()* e *recv()*, que identificam a existência de um canal aberto de entrada ou saída e passam então a receber e transferir dados de e para os mesmos, conforme demostrado na FIGURA 23, da seguinte forma:

1) A função *send()*, no início de sua operação, verifica se os canais foram ativados. Caso estejam, os dados serão lidos dos mesmos para então serem enviados. Caso não seja detectado a abertura de um canal, os dados serão enviados normalmente via *socket*;
2) A função *recv()*, da mesma forma como ocorre na função *send()*, verifica inicialmente se os canais foram abertos. Em caso positivo, os dados serão escritos no *pipe* correspondente até o término da operação de recepção. Caso não exista nenhum canal aberto, os dados serão recebidos normalmente via *socket* conforme a definição original da função.

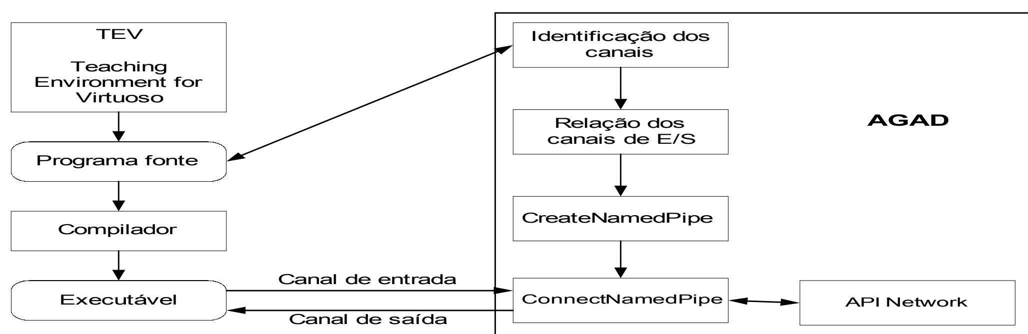


FIGURA 22 – Interação entre as Aplicações do TEV e do AGAD

Os procedimentos adotados nas funções *send()* e *recv()* foram extendidas para todas as funções da API *Network,* que foi utilizada no AGAD. No entanto, essas alterações foram implementadas de maneira que a mesma possa ser utilizada tanto em um ambiente integrado com o TEV como também em aplicações geradas e executadas a partir do AGAD.

5.6 Resumo do Capítulo

Neste capítulo foram detalhados os procedimentos necessários para integrar o AGAD (Ambiente para Geração de Aplicações Distribuídas) com o TEV (Ambiente para Desenvolvimento de Aplicações Paralelas), para permitir a interação de uma aplicação de tempo real que utiliza canais para comunicação entre suas tarefas, com uma aplicação baseada no paradigma cliente/servidor que utiliza o mecanismo IPC *Socket* em sua implementação. Para isso foram abordados as funcionalidades do TEV e como o AGAD, através de um algoritmo de seleção, identifica os canais, para em seguida estabelecer a conexão entre as aplicações.

Capítulo 6
UTILIZANDO O AGAD NA GERAÇÃO DE UMA APLICAÇÃO INTEGRADA AO TEV

6.1 Gerando Aplicações AGAD Integradas ao TEV

Para elucidar a utilização da ferramenta AGAD e sua integração com o TEV serão gerados, a partir das mesmas, duas aplicações, uma para atuar como Servidor e outra como Cliente.

6.2 Implementando uma Aplicação Servidor

A seguir serão descritos os passos necessários para a implementação de um Servidor:

1) Inicialização da *Winsock* do Windows 2000 ou NT com a versão 2 a partir da utilização das funções "Carregar Winsock 2" e "*WSAStartup*";
2) Criação de um *Socket* para ser o descritor do processo de transferência de dados. Para isto, será utilizado a função *Socket* do sub-menu *Socket*, alterando seus parâmetros para a criação do mesmo fazendo uso do TCP como protocolo de transporte;
3) Com o *socket* criado, será acionada a função *Bind* para relacionar o *Socket* com um endereço IP, neste caso 10.1.100.10, e a uma porta local, neste caso com número 2000;
4) Através da opção *Listen*, o *socket* será inicializado, ficando a aplicação pronta para o comando *Accept*, que irá colocar Servidor no modo de espera, aguardando a conexão de uma aplicação Cliente.

Na FIGURA 16, estão exibidas todas as mensagens que foram geradas a partir dos passos acima. Existe a possibilidade de configurar a ferramenta para adicionar na mensagem a hora na qual a mesma foi mostrada ao usuário, obtendo-se assim, uma ordem cronológica das execuções.

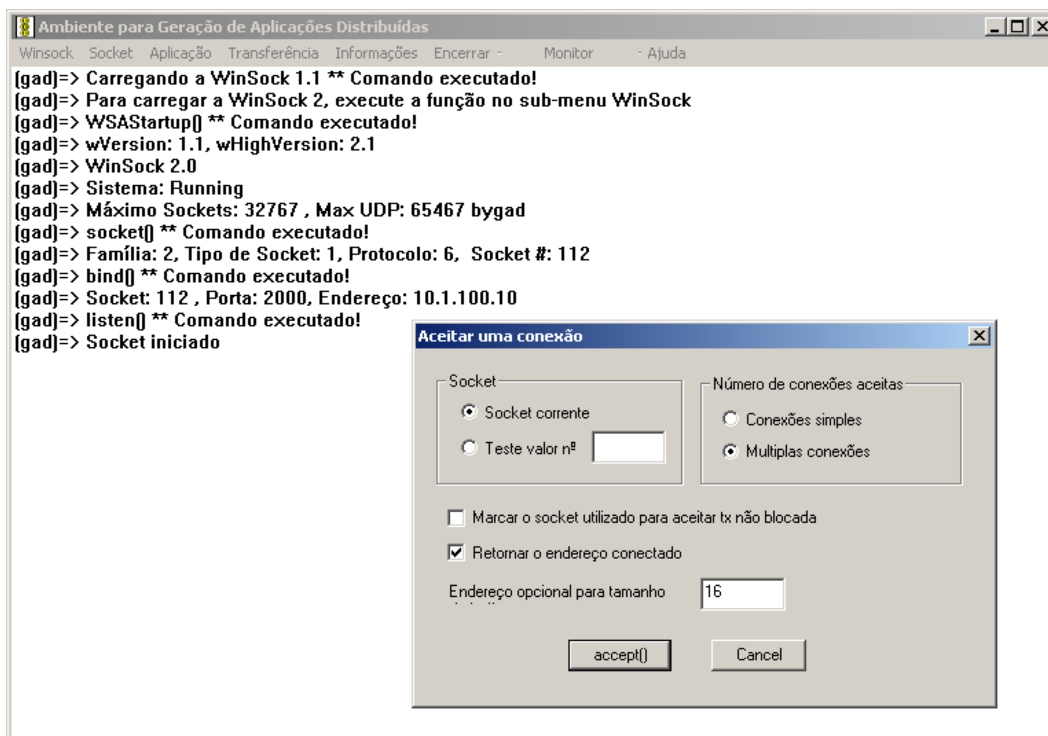


FIGURA 23 – Mensagens Exibidas pela Ferramenta na Criação da Aplicação Servidor

6.2.1 Efetuando a conexão do Servidor Socket com uma Aplicação do TEV

Após o servidor *socket* ter sido elaborado a partir do AGAD, será necessário efetuar a conexão do mesmo com uma aplicação desenvolvida no TEV, neste exemplo um receptor simples de mensagens denominado RECEPTOR. Para tanto, faz-se necessário a utilização da opção “Canais” no menu “Winsock” do AGAD para selecionar o arquivo *tasks.c* do diretório onde se encontram os fontes da aplicação RECEPTOR, para com isto identificar os nomes dos canais de entrada e saída que estão sendo utilizados. Esta operação pode ser visualizada na FIGURA 23.

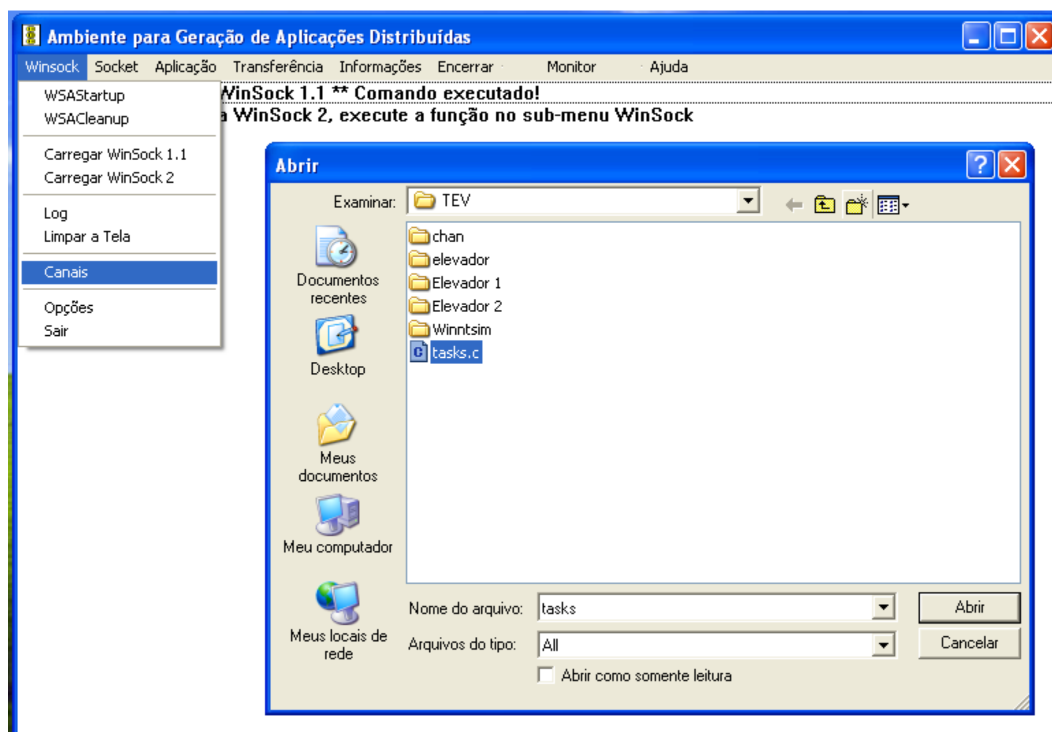


FIGURA 24 – Seleção de canais de entrada e saída em aplicação desenvolvida no TEV

Após a operação de seleção de canais, a aplicação Servidor do AGAD ira encaminhar toda mensagem que for recepcionada de uma aplicação Cliente do AGAD para a aplicação RECEPTOR do TEV. Esta por sua vez poderá dar o tratamento que for necessário a esta mensagem de acordo com o que for estipulado previamente pelo desenvolvedor.

6.3 Implementando uma Aplicação Cliente

Uma aplicação Cliente irá se comunicar via *Socket* com o Servidor, para tanto, os seguintes passos são necessários para a sua implementação:

1) Os dois primeiros passos são idênticos aos dois primeiros da criação do Servidor;
2) Com o *socket* criado, será acionada a função *Bind*, para relacioná-lo com o endereço IP 127.0.0.1 e com a porta local número 1000;

3) A partir da inicialização do *socket*, a aplicação Cliente está pronta para a função *Connect*, que irá contactar o Servidor a partir dos parâmetros que forem especificados.

No momento em que a função *Connect* for iniciada, irá surgir no Servidor a mensagem confirmando a conexão. A partir deste ponto, o usuário poderá utilizar as funções para transmissão ou recepção de dados, bem como utilizar as várias funções existentes para a obtenção de informações das duas aplicações, como situação da conexão através das portas, nome dos computadores envolvidos, entre outras.

6.4 Resumo do Capítulo

Neste capítulo foi descrito como podem ser geradas aplicações Cliente/Servidor a partir do AGAD e como efetuar a conexão das mesmas com uma aplicação desenvolvida no TEV utilizado recursos de *sockets* e *pipes*. As aplicações geradas podem ser executadas em um único microcomputador ou então em microcomputadores diferentes interligados em uma estrutura de rede.

Capítulo 7
CONSIDERAÇÕES FINAIS

7.1 Conclusão

Os estudos realizados durante a realização deste trabalho, visando o desenvolvimento de um utilitário que auxiliasse no desenvolvimento de aplicações distribuídas, resultaram na implementação de uma biblioteca de classes contento funcionalidades que permitem seu uso na geração de sistemas baseados no paradigma Cliente/Servidor. Esta biblioteca, denominada API *Network*, foi incorporada ao utilitário chamado AGAD, que visa facilitar o uso destas classes através de uma interface gráfica, oferecendo uma série de opções para a geração de sistemas distribuídos em uma rede de computadores. A implementação desta ferramenta foi modelada de forma que as funções necessárias para o desenvolvimento destes aplicativos, possam ser executadas de forma interativa conforme forem selecionadas pelo projetista. Esta funcionalidade permite que alterações ou erros provocados por seqüências inválidas ou parâmetros indevidos na utilização das funções, possam ser identificados e resolvidos no momento da ocorrência. Isto incorpora à ferramenta AGAD uma característica muito importante, que é a possibilidade de sua utilização como auxiliar no ensino das técnicas e mecanismos que compreendem o desenvolvimento deste tipo de aplicação. Sendo este o enfoque destinado à ferramenta neste trabalho, foi incorporado à mesma, um monitor de tráfego que através de funções especificas capturam os pacotes originados na transferência de dados pelo AGAD, nas conexões via *socket*. Buscou-se com isto, deixar claro os detalhes existente durante o processo de comunicação, fornecendo ao usuário meios para ele possa verificar se a intenção existente durante o processo de construção foi alcançada após a execução da sua aplicação.

Outra funcionalidade importante adicionada ao AGAD, foi a comunicação do mesmo com as aplicações geradas a partir do TEV (*Teaching Enviroment for Virtuoso*), que originalmente não contemplava a execução destas aplicações distribuídas em uma rede de computadores. Esta integração veio

aumentar as possibilidades de uso das mesmas, pois demonstram em sua utilização, conceitos que envolvem diversos mecanismos de comunicação inter-processos, como *pipes,* transferência de mensagens, *sockets* entre outros. O modelo integrador que foi implementado permite ainda, que essas duas ferramentas possam ser utilizadas em conjunto ou em separado, dependendo das necessidades para as quais forem solicitadas, dando a cada uma delas, uma independência funcional muito importante se forem utilizadas como auxiliares no ensino dos conceitos mencionados.

7.2 Trabalhos Futuros

Apesar da modelagem do AGAD e sua interação com o TEV estar concluída, podem ser incorporadas à implementação deste modelo uma série de melhorias adicionais, tais como:

1) Implementar um editor para permitir a visualização e manutenção do arquivo fonte gerado pelo AGAD, que contém a seqüência de procedimentos escolhidos durante a construção da aplicação cliente/servidor;
2) Implementar no AGAD uma interface gráfica mais amigável, seguindo a ergonomia adotada na construção do TEV, buscando assim uma padronização na aparência das duas ferramentas;
3) Incorporar no TEV a API Network como mais uma de suas funcionalidades, fornecendo a esta ferramenta, os mecanismos necessários para a utilização das funções que são exigidas na construção de aplicações distribuídas em uma rede de computadores.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] WILLIS, Barry. **Distance education at a Glance** (1996) Series of Guides prepared by Emgineering Outreach at the University of Idaho. URL: http://www.uidaho.edu/evo/distglan.html. (acessado em 18.10.2001)

[2] COMER, Douglas E; STEVENS, David L. **Interligação Em Rede Com Tcp/Ip – Vol. I.** Rio de Janeiro: Editora Campus, 1998.

[3] COMER, Douglas E; STEVENS, David L . **Internetworking with TCP/IP : client - server programming and applications : windows sockets version.** New Jersey: Prentice-Hall, 1997.

[4] COMER, Douglas E; STEVENS, David L . **Interligação Em Rede Com Tcp/Ip - Vol. II.** Rio de Janeiro: Editora Campus, 1999.

[5] STALLINGS, William. **Arquitetura e Organização de Computadores**. 5Ed. São Paulo: Editora Prentice Hall, 2002.

[6] TANENBAUM, Andrew S . **Redes de computadores**. 4 ed. Rio de Janeiro: Editora Campus, 1999.

[7] TANENBAUM, A.S. **Distribuited Operating Systems**. Prentice Hall Press, 1995.

[8] KOYMANS, R. **Specifyng Real Time Properties with Metric Temporal Logic**. November 1990. Real Time Systems Journal, v.2, n.4, p. 255-299, 1990.

[9] BOTELHO, Tomás de Aquino Tinoco. **Análise de Desempenho da Arquitetura Cliente/Servidor Orientada a Objeto.** Tese de Mestrado, IME, Dezembro/1995.

[10] MORON, C.E., Ribeiro, J.R.P., Silva, N.C. **A Teaching Environment for the Development of Parallel Real-Time Programs**. In: Frontiers in Education'98, 1., 1998, Tempe, Arizona - EUA. EP Innovations, 1998.

[11] RIBEIRO, J.R.P., Silva, N.C., Moron, C.E. **A Visual Environment for the Development of Parallel Real-Time Programs**. In: Lecture Notes in

Computer Science, v. 1388, p. 994-1014, 1998.

[12] SOARES, Luiz Fernando Gomes; LEMOS, Guido; COLCHER, Sérgio . **Redes de computadores : das Lans, Mans e Wans às Redes ATM**. 2 ed. Rio de Janeiro: Campus, 1999. 705 p.

[13] **Virtuoso – The Virtual Single Processor Programming System. User Manual**, Version 4.0, Eonic System, Inc.

[14] RICHTER, Jeffrey . **Windows avançado : guia de desenvolvimento para API WIN32, para Windows NT e Windows 95.** São Paulo : Makron Books, 1995.

[15] RICCIONI, Paulo Roberto. **Introdução a Objetos Distribuídos com Corba**. Florianópolis: BookStore Livraria Ltda, 2000.

[16] QUATRANI, T. **Visual Modeling with Rational Rose and UML**. Addison-Wesley, 1998.

[17] TANENBAUM, Andrew S., WOODHUL, Albert S.. **Sistemas operacionais.** 2ed. Porto Alegre: Bookman, 2000.

[18] BRESLAU, L. et. al. **Advances in Network Simulations**. Computer, Maio 2000.

[19] SIMON, R.J. **Windows NT Win32 API Superbible**. Waite Group Press, 1997.

[20] HOLZNER, Steven. **Programando Visual C++ 6**. Rio de Janeiro: Makron Books, 1999.

[21] LARMAN, Craig. **Utilizando UML e padrões**. Porto Alegre: Bookman, 2000. 492p.

[22] SIYAN, K.S., ET AL. **Windows NT Server 4**. New Riders, Second Edition, 1997.