

**UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO**

***MÉTODO PARA DESENVOLVER AGENTES
ADAPTATIVOS EM GERÊNCIA DE REDES USANDO
REDES NEURAIS***

Elvis Melo Vieira

Carlos Becker Westphall
Orientador

Dissertação submetida à Universidade Federal de Santa Catarina para obtenção de grau de
MESTRE EM CIÊNCIA DA COMPUTAÇÃO.

Florianópolis, Fevereiro de 1997

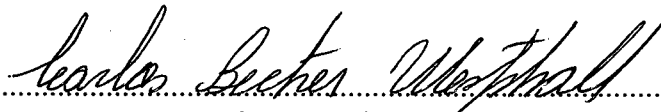
**MÉTODO PARA DESENVOLVER AGENTES
ADAPTATIVOS EM GERÊNCIA DE REDES USANDO
REDES NEURAIIS**

Elvis Melo Vieira

Esta dissertação foi julgada adequada para a obtenção do título de

MESTRE EM CIÊNCIA DA COMPUTAÇÃO

Especialidade **SISTEMAS DE COMPUTAÇÃO** e aprovada em sua forma final pelo
programa de **PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO**.



Prof. Dr. Carlos Becker Westphall


Orientador



Prof. Dr. Murilo Silva de Camargo

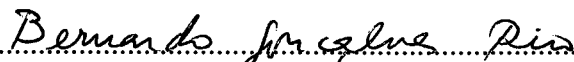
Coordenador do Curso de PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

BANCA EXAMINADORA:



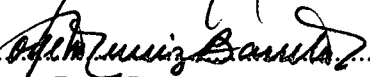
Prof. Dr. Carlos Becker Westphall

Presidente



Prof. Dr. Bernardo Gonçalves Riso


Prof. Dr. Manoel Camillo de Oliveira Penna Neto



Prof. Dr. Jorge Muniz Barreto

AGRADECIMENTOS

É com grande estima que presto agradecimentos a todas as pessoas que direta ou indiretamente auxiliaram no desenvolvimento deste trabalho.

Agradecimentos ao Prof. Carlos Becker Westphall pela confiança depositada, o apoio dedicado e, principalmente, pelo valoroso trabalho de orientação.

Agradecimentos especiais a Solange Teresinha Sari que auxiliou de duas maneiras: tecnicamente, sendo fonte de conhecimentos e na ajuda da elaboração do trabalho, e como esposa, me animando e me dando muita confiança.

Aos professores do programa, meu profundo agradecimento aos conhecimentos transmitidos. À coordenação do curso de pós-graduação agradecimentos pelos excelentes serviços prestados.

Agradecimentos ao NPD da UFSC, onde encontrei o ambiente propício, principalmente através da união e amizade dos seus funcionários. Em especial ao pessoal de Gerência de Redes e principalmente, do Laboratório de Interoperabilidade de Redes onde este trabalho foi desenvolvido, que na pessoa de seu coordenador Edison Tadeu Lopes Melo, tiveram muita paciência e dedicação para comigo.

Agradecimentos aos meus pais, Antônio Frederico Vieira e Veralba Melo Vieira e ao meu irmão, Antonio Carlos Melo Vieira, pelo incentivo e apoio aos estudos durante todos estes anos.

ÍNDICE

AGRADECIMENTOS	3
ÍNDICE	i
ÍNDICE DE FIGURAS	iv
ÍNDICE DE QUADROS	vi
RESUMO	vii
ABSTRACT	viii
1. INTRODUÇÃO	1
1.1 Objetivos	1
1.2 Justificativa	2
1.3 Escopo e Limitações	2
1.4 Organização do Trabalho	3
2. GERÊNCIA DE REDES	5
2.1 Características Desejáveis da Atividade de Gerência	5
2.2 Modelos de Gerência	6
2.3 Representação de Dados	8
2.4 Axioma Fundamental do Gerenciamento de Rede Internet	9
2.5 Estrutura da Informação de Gerenciamento	10
2.6 Protocolo de Gerenciamento de Rede Internet	15
2.7 SMUX	21
3. REDES NEURAIS ARTIFICIAIS	25
3.1 Soluções Algorítmicas	25
3.2 Características das Redes Neurais	26
3.3 Neurônio Artificial	27
3.4 Função de Ativação	28
3.5 Redes Neurais Artificiais	29
3.6 Treinamento em Redes Neurais	30
3.7 Realimentação de Redes Neurais	32
3.8 <i>Backpropagation</i>	34
4. MODELAGEM E PROJETO ORIENTADO A OBJETOS	40
4.1 Módulos como Classes de Objetos	40
4.2 Modelagem e Projeto Orientado a Objetos	41

4.3 Modelo Estático	41
4.4 Diagrama de Estados	45
4.5 Diagrama de Fluxo de Dados	46
5. AGENTE ADAPTATIVO	49
5.1 Modelos de Gerência Não-Adaptativos	49
5.2 Implementação da Adaptação ao Modelo de Gerência	50
5.3 Extensão Adaptativa de um Agente	51
6. AMBIENTE PARA O DESENVOLVIMENTO DE AGENTES ADAPTATIVOS (ADAA)	56
6.1 Domínio do Problema	56
6.2 Objetivos	58
6.3 Descrição do Ambiente	59
6.4 SNMPlib	60
6.5 Neurolib	82
7. BIBLIOTECA DE CLASSES DE GERÊNCIA DE REDES - SNMPLIB	90
7.1 Diagrama de Classes	90
7.2 <i>sAgent</i>	91
7.3 <i>sMO</i>	101
7.4 <i>sOT</i>	104
7.5 <i>sOS</i>	106
7.6 <i>sOID</i>	107
7.7 <i>sModuleMIB</i>	110
7.8 <i>sSMUX</i>	110
7.9 <i>sTE</i>	115
7.10 <i>sVarBind</i>	115
7.11 <i>sPDU</i>	115
7.12 <i>sTrap</i>	117
7.13 <i>sPending</i>	119
8. BIBLIOTECA DE CLASSES DE REDES NEURAIIS - NEUROLIB	120
8.1 Hierarquia de Classes	120
8.2 <i>nNet</i>	121
8.3 <i>nNSuperv</i>	124
8.4 <i>nSuperv</i>	124
8.5 <i>nBP</i>	128
9. PROGRAMA PARA PROTOTIPAÇÃO E TREINAMENTO DE REDES NEURAIIS - NEUROPROTO	133
9.1 Descrição do NeuroProto	133

9.2 Processo de Definição de uma Rede Neural _____	136
9.3 Processo de Treinamento _____	137
9.4 Processo de Teste _____	137
9.5 Exemplo de uma Prototipação _____	138
10. CONCLUSÃO _____	146
10.1 Integração das Tecnologias de Suporte _____	146
10.2 Desenvolvimento das Classes Básicas e do Programa de Prototipação e Treinamento _____	148
10.3 Visão de Usuário _____	149
10.4 Alcance dos Objetivos _____	150
10.5 Dificuldades Encontradas _____	151
10.6 Aplicabilidade de Agentes Adaptativos _____	151
10.7 Perspectivas Futuras _____	152
ANEXO A : DIAGRAMAS DA BIBLIOTECA SNMPLIB _____	154
ANEXO B: DIAGRAMAS DA BLIBLIOTECA NEUROLIB _____	169
ANEXO C: ARQUIVOS INCLUDES PARA C++ DA SNMPLIB _____	174
ANEXO D : ARQUIVOS INCLUDES PARA C ++ DA NEUROLIB _____	185
BIBLIOGRAFIA _____	192
GLOSSÁRIO _____	195

ÍNDICE DE FIGURAS

Figura 2-1 Modelo de Gerência	7
Figura 2-2 Especificação em ASN.1 da estrutura "Aluno"	9
Figura 2-3 Definição da Macro OBJECT TYPE	12
Figura 2-4 Subárvore da Internet	13
Figura 2-5 Definição de list e table	14
Figura 2-6 Representação da interação de um pedido SNMP	15
Figura 2-7 Representação da interação de um trap SNMP	16
Figura 2-8 - Mensagem SNMP	16
Figura 2-9 Protocolo Multiplexador do SNMP - SMUX	21
Figura 2-10 Estabelecimento e encerramento de uma associação SMUX	22
Figura 2-11 Comunicação SNMP de um subagente	23
Figura 3-1 Neurônio Artificial	28
Figura 3-2 Rede Neural de Uma Camada	29
Figura 3-3 Rede Neural de Múltiplas Camadas	31
Figura 3-4 Redes Feedforward	33
Figura 3-5 Redes FeedBack	34
Figura 3-6 Neurônio Fundamental do Backpropagation	35
Figura 3-7 Redes Neurais de Múltiplas Camadas Treináveis pelo Backpropagation	35
Figura 3-8 Ajustes dos Pesos	38
Figura 4-1 Notação de uma Classe no Diagrama de Classes	42
Figura 4-2 Classes Pessoa e Arquivo	42
Figura 4-3 Relação de Herança Entre as Classes Pessoa, Aluno e Professor	43
Figura 4-4 Multiplicidade de Associações	44
Figura 4-5 Associação trabalha-para	44
Figura 4-6 Notação da Relação de Agregação	45
Figura 4-7 Exemplo de Agregação	45
Figura 4-8 Notação de Estados e Eventos	46
Figura 4-9 Diagrama de Estados de um Teclado Numérico	46
Figura 4-10 Representação de Processos, Atores e Depósito de dados	47
Figura 4-11 Processo da Divisão de um Número Inteiro	47
Figura 4-12 Diagrama de Fluxo de Dados da Tabela Periódica	48
Figura 5-1 Modelo de Gerência	50
Figura 5-2 Um Subagente Adaptativo	52
Figura 5-3 Implementação do Agente Adaptativo com Redes Neurais	54
Figura 6-1 Modelo Agente-Gerente com Subagente	60
Figura 6-2 Fluxo da Operação SNMP Get em uma Instância de um Subagente	63
Figura 6-3 Classes route e IP	65
Figura 6-4 Definição na MIB-II das Classes IP e route e da Agregação routeTable	66
Figura 6-5 Diagrama de Classes	69
Figura 6-6 Diagrama de Estados para os Processos ProcGet e ProcSet	70
Figura 6-7 Diagrama de Fluxo de Dados para os Processos Run, ProcGet e ProcSet	71
Classe procMO	71
Figura 6-8 Diagrama de Estados da Classe procMO	72
Figura 6-9 Diagrama de Fluxo de Dados para a Classe procMO	73
Figura 6-10 Arquivo process.h	74
Figura 6-11 Arquivo process.cpp (parte 1)	75
Figura 6-12 Arquivo process.cpp (parte 2)	76
Figura 6-13 Arquivo procmo.h	77
Figura 6-14 Arquivo procmo.cpp (part1)	78
Figura 6-15 Arquivo procmo.cpp (parte 2)	79
Figura 6-16 Arquivo procmond.h	80

Figura 6-17 Arquivo <i>procmond.cpp</i>	81
Figura 6-18 Representação da Subárvore Privada Controlada por <i>procmond</i>	82
Figura 6-19 Classe Interface IEEE 802.3	84
Figura 6-20 Arquivo <i>ieee8023.h</i>	86
Figura 6-21 Arquivo <i>ieee8023.cpp</i>	86
Figura 6-22 Arquivo <i>main_iee8023.cpp</i>	87
Figura 6-23 Arquivo <i>iee8023.def</i>	88
Figura 6-24 Classe Interface IEEE 802.3	89
Figura 7-1 Diagrama de Classes de Gerência de Redes	91
Figura 7-2 Descrição das Partes de um Subagente	92
Figura 7-3 Estado DISPATCH do Subagente	93
Figura 7-4 Processo Dispatch	94
Figura 7-5 Estado GETMESSAGE	97
Figura 7-6 Método <i>procGet</i> de uma Instância SA da Classe <i>sSagent</i>	98
Figura 7-7 Método <i>procSet</i> do Subagente	100
Figura 7-8 Método <i>run</i> da Classe Subagente	101
Figura 7-9 Método <i>procGet</i> da Classe Objeto Gerenciável	102
Figura 7-10 Método <i>procSet</i> da Classe Objeto Gerenciável	103
Figura 7-11 Construtor e método <i>getListOT</i> da Classe Objeto Gerenciável	104
Figura 7-12 Macro ASN-1 <i>Objet-Type</i>	105
Figura 7-13 Diagrama de Estado da Classe <i>sSMUX</i>	114
Figura 8-1 Diagrama de Classes da Biblioteca de Redes Neurais	123
Figura 8-2 Implementação do Método <i>activation</i>	124
Figura 8-3 Diagrama de Estados para a Classe <i>Superv</i>	125
Figura 8-4 Diagrama de Fluxo de Dados para a Classe <i>nSuperv</i>	126
Figura 8-5 Diagrama de Fluxo de Dados do Processo <i>train</i>	127
Figura 8-6 Diagrama de Estados do Estado TRAIN	127
Figura 8-7 Topologia do Backpropagation	129
Figura 8-8 Diagrama de Estados Figura 8-9 Diagrama de Fluxo de Dados	130
Figura 8-10 Diagrama de Estados dos Estados FORWARD E BACKWARD	131
Figura 8-11 Diagrama de Fluxo de Dados do processo forward	131
Figura 8-12 Diagrama de Fluxo de Dados do Processo Backward	132
Figura 9-1 Interface do NeuroProto	135
Figura 9-2 Componentes do Sistema de Acesso Remoto Discado	139
Figura 9-3 Pré-processamento	141
Figura 9-4 Modelo de classes do agente adaptativo	145

ÍNDICE DE QUADROS

<i>Quadro 2-1 Tipos definidos pela SMI da Internet</i> _____	14
<i>Quadro 5-1 Características da Implementação da Adaptação ao Modelo Tradicional</i> _____	51
<i>Quadro 6-1 Correspondência entre Atributos Primitivos e Tipos Simples e de Aplicação</i> _____	65
<i>Quadro 6-2 Códigos de Diagnósticos</i> _____	85
<i>Quadro 7-1 Classes que Implementam Tipos Pré-Definidos na SMI</i> _____	107
<i>Quadro 9-1 Parâmetros de Definição da Rede Neural</i> _____	137
<i>Quadro 9-2 Intervalos de Valores Resultantes do Pré-Processamento</i> _____	143
<i>Quadro 9-3 Resultados Obtidos de Vários Conjuntos de Treinamento</i> _____	144
<i>Quadro 9-4 Resultados do Procedimento de Teste</i> _____	144

RESUMO

Um agente adaptativo é uma extensão ao modelo agente-gerente usado pela maioria dos padrões de gerência como o OSI e o Internet. A proposta é estender o agente SNMP do padrão Internet com a criação de um subagente para controlar objetos gerenciáveis. O subagente e o agente comunicam-se através do protocolo SMUX da Internet. Entretanto, o subagente possui características adaptativas implementadas, como é sugerido, através de redes neurais. Desta forma o novo agente, formado a partir desta extensão, pode reagir a possíveis falhas ou problemas notificados pelos objetos gerenciáveis sob seu controle. As ações ou reações são fornecidas pela implementação da característica adaptativa do agente. O trabalho é composto da definição do agente adaptativo e da descrição detalhada de um suporte para prover a sua construção. Para o desenvolvimento deste suporte, foi utilizada uma metodologia de análise e projeto orientada a objetos denominada OMT (*Object Modeling Technique*). O resultado é a modelagem de duas bibliotecas, uma contendo classes para gerência de redes e a outra, classes de redes neurais. Além dessas duas bibliotecas, é implementado um programa de prototipação e treinamento, onde o implementador pode validar a solução via agente adaptativo utilizando redes neurais. Isto é necessário, pois nem sempre uma rede neural é a melhor solução. Outras técnicas podem ser empregadas, obtendo-se resultados mais adequados, como sistemas especialistas e sistemas evolucionistas.

ABSTRACT

An adaptative agent is an extension of the manager-agent model used by the majority of management specification as the OSI and Internet. The propose is to extend the SNMP agent of the Internet model creating a subagent to control managed objects. The subagent and the agent communicate through the Internet SMUX protocol. Although, the subagent has adaptative features implemented, as suggested, through neural networks. This way, the new agent created from this extension, can react to possible failures or problems notificated by the managed objects under by its control. The actions or reactions are offered by implementation of the agent adaptative feature. The work is composed of the adaptative agent definition and the detailed description of a support to offer its construction. For the design of this support, a methodology of object oriented analisys and design was used, called OMT (Object Modeling Technique). The result is the model of two libraries, one having classes to network management and the other neural networks classes. Besides these two libraries, is implemented a prototype and training program, where the implementation can validate the solution by adaptative agent using neural networks. This is necessary, because not aways a neural network is the best solution. Other techniques can be used, obtaining more suitable adequated results, as specialist systems and evolutionist systems.

1. INTRODUÇÃO

O modelo tradicional de gerência de redes, conhecido como modelo agente-gerente, fornece uma gerência onde qualquer decisão para reagir a um problema notificado por um agente, tem que ser tomada pelo operador humano que manipula a console do sistema de gerência.

A idéia de agentes adaptativos situa-se em uma tentativa de prover a característica adaptativa aos agentes que controlam objetos gerenciáveis no âmbito da gerência de redes de computadores. Os agentes assim construídos, podem reagir aos problemas encontrados nos objetos gerenciáveis, tentando manter a normalidade e as características operacionais da rede.

Desta forma, a construção de agentes adaptativos pode mostrar-se muito preciosa e de aplicação imediata, visto que existem muitos esforços no sentido do desenvolvimento da gerência de redes automatizada. Este tipo de gerência procura construir sistemas autocontroláveis, tentando evitar a necessidade da intervenção humana.

1.1 Objetivos

Objetivo geral é construir um ambiente de desenvolvimento de agentes adaptativos aplicados em gerência de redes de computadores.

Para a construção do ambiente de desenvolvimento, os seguintes objetivos específicos devem ser alcançados:

- Modelar o ambiente utilizando uma metodologia de desenvolvimento *desoftware*.
- Viabilizar uma metodologia de desenvolvimento para o usuário do ambiente.
- Disponibilizar um programa de prototipação e treinamento que ajude a analisar a aplicabilidade do uso de redes neurais em um agente adaptativo para solucionar um determinado problema.

- Construir um conjunto de módulos, comuns a todos os agentes adaptativos, que implementem conceitos de gerência de redes internet.
- Construir um conjunto de módulos, comuns a todos os agentes adaptativos, que implementem conceitos de redes neurais.
- Avaliar a aplicabilidade do ambiente em situações reais.

1.2 Justificativa

A construção de agentes adaptativos apresentada neste trabalho, utiliza redes neurais artificiais para prover a característica de adaptação. Neste sentido, o agente proposto tem a capacidade de reagir a possíveis problemas que não estejam colocados em uma base de conhecimento, mas que diferem destes com pequenas variações.

Com isto, consegue-se construir um agente, guardando as devidas limitações, que atuaria semelhantemente ao operador humano. Portanto, certos problemas não solucionáveis algoritmicamente ou que apresentam soluções inviáveis ou ineficientes, podem encontrar no agente adaptativo, uma forma alternativa que pode-se mostrar mais adequada ou eficiente.

1.3 Escopo e Limitações

O trabalho proposto é utilizado na gerência automatizada de recursos computacionais em redes de computadores internet como por exemplo, para o controle de processos de um sistema operacional, o monitoramento de interfaces de comunicação ou o controle de tabela de rotas e tráfego em uma rede.

Para o desenvolvimento deste trabalho foi utilizada a metodologia orientada a objetos denominada OMT (*Object Modeling Technique*), tanto para a fase de análise, como para a fase de projeto.

A implementação foi realizada no sistema UNIX System V, utilizando o ambiente gráfico CDE (*Common Desktop Environment*) e como linguagem de programação, o C++.

A parte adaptativa dos agentes desenvolvidos é limitada ao emprego de uma rede neural com treinamento *Backpropagation*. Entretanto, outras redes podem ser empregadas dependendo do domínio da aplicação.

1.4 Organização do Trabalho

O trabalho apresentado é organizado em cinco partes principais:

- A primeira parte engloba a introdução, compreendendo o capítulo 1. O objetivo é dar uma visão geral do escopo, da proposta e as limitações do trabalho.
- A segunda parte é a revisão bibliográfica, compreendendo os capítulos 2, 3 e 4, onde o objetivo é fornecer um embasamento teórico necessário para a compreensão do texto. No capítulo 2 são descritos conceitos básicos de gerência de redes de computadores, com ênfase no padrão Internet. No capítulo 3 são descritos conceitos gerais de redes neurais, sem a pretensão de fornecer uma abordagem profunda no assunto. No capítulo 4, é introduzida a metodologia OMT, amplamente usada para estruturar o projeto final.
- A terceira parte abrange os capítulos 5 a 9, onde é descrito o trabalho desenvolvido. Os capítulos 5 e 6 desenvolvem a idéia do agente adaptativo. No capítulo 5 é proposto a inclusão da característica adaptativa ao modelo tradicional de gerência de redes, relatando suas vantagens e desvantagens. Neste capítulo, também são discutidas algumas formas de implementações da característica adaptativa, dando ênfase ao agente adaptativo. No capítulo 6 é descrito genericamente, um ambiente para o desenvolvimento de agentes adaptativos e sua utilização por parte do implementador. Os capítulos 7, 8 e 9 discutem com profundidade cada uma das partes do ambiente. No capítulo 7 são discutidos os componentes da Biblioteca de Classes de Gerência de Redes (SNMPLib), no capítulo 8 são discutidos os componentes da Biblioteca de

Classes de Redes Neurais (Neurolib) e, no capítulo 9, é descrito o Programa para Prototipação e Treinamento de Redes Neurais (NeuroProto).

- A quarta parte envolve o capítulo 10. Neste capítulo é feito um resumo do trabalho proposto, verificando se os objetivos iniciais foram atingidos e, principalmente, relacionando as dificuldades encontradas. Por fim, é discutida a aplicabilidade de agentes adaptativos na gerência de redes, tirando-se algumas conclusões importantes.
- Concluindo o trabalho, na quinta e última parte são colocados os anexos e as referências bibliográficas: os dois primeiros anexos apresentam os diagramas OMT da SNMPLib e da Neurolib. Os dois últimos, os arquivos *header* (*includes *.h*) das classes da SNMPLib e Neurolib, descrevendo a interface pública em C++ das suas classes. Por último, a referência bibliográfica lista obras que podem ser pesquisadas para um melhor detalhamento dos temas abordados em várias partes do trabalho.

2. GERÊNCIA DE REDES

O crescimento do tamanho das redes, do seu número de usuários e dos serviços disponíveis, torna a gerência operacional destas, um elemento dinâmico e vital para o planejamento de outras áreas da informática.

Com o dinamismo existente em uma rede de computadores, a equipe de administração e gerência deve estar preparada para solucionar os constantes problemas que muitas vezes são inevitáveis. Uma gerência operacional eficaz é imprescindível para que as falhas sejam solucionadas o mais rápido possível, preferencialmente, sem que o usuário sinta alguma anormalidade.

Em função destes e de outros fatores, gerenciar uma rede significa manter sob controle todo o seu funcionamento, abrangendo entre outras atividades, o controle da configuração da rede, o controle do acesso aos recursos compartilhados, a manutenção do tráfego e a disponibilidade dos serviços em níveis aceitáveis.

2.1 Características Desejáveis da Atividade de Gerência

As características desejáveis de uma atividade de gerência variam com as necessidades das instituições, mas algumas podem ser enumeradas:

1. **Eficiência:** Se caracteriza pela tomada de decisões corretas, rápidas e eficazes. Neste caso, a tomada de uma decisão visando sanar ou evitar um problema pode ou não ter efeito. Em geral, equipes de gerência bem treinadas e, principalmente experientes, tendem a ser mais eficientes.

2. **Continuidade** : Se caracteriza pela não interrupção dos serviços, mesmo quando está sendo reparada alguma falha. Aqui, o objetivo principal é procurar a disponibilidade máxima de todos os serviços, principalmente nos dias atuais, com a competitividade acirrada entre as empresas.
3. **Pró-Ativa**: Provê uma espécie de controle de qualidade da rede. Ela difere das primeiras, pois atua no sentido de que prevenir é melhor que remediar. Portanto, o objetivo é procurar prever o acontecimento de possíveis problemas e agir de modo a evitar que eles ocorram. Um exemplo que pode ser dado de gerência pró-ativa está associado com o gargalo no tráfego de um roteador. A gerência eficiente das rotas, filtragem de pacotes, monitoração do tráfego e alocação adequada de linhas de comunicação são algumas medidas que uma gerência pró-ativa pode considerar para evitar o problema de congestionamento em um roteador.
4. **Amplitude do Domínio**: Se caracteriza pela abrangência do seu domínio de gerência, geralmente indicando um domínio administrativo como uma subrede local ou uma rede de longa distância. Um bom seria a rede de um campus universitário, onde os vários setores podem possuir redes com administração própria, sendo que cada administrador somente pode atuar nas redes que estão dentro do seu domínio de gerência.

O grau de importância de cada uma destas características ou a inclusão de outras é ditada pela política de gerência local, a qual deve ser bem definida para uma gerência eficaz.

2.2 Modelos de Gerência

Os modelos de gerência de redes atuais são produtos de estudos e pesquisas realizados por universidades, indústrias e organismos de padronização. Um modelo de gerência visa organizar e montar uma estrutura através da qual, a atividade de gerência possa ser realizada [19][20]:

Um modelo de gerência de redes é caracterizado pelos seguintes componentes [19][20], representados pela figura 2.1:

1. Um ou mais **nós gerenciáveis**, cada um contendo um agente que responde a pedidos ou envia notificações ao(s) gerente(s).

2. No mínimo uma **estação de gerenciamento** de rede (ou *NMS - Network Management Station*), executando uma (ou mais) aplicação de gerenciamento, frequentemente denominada de **gerente**, o qual envia pedidos e recebe respostas ou notificações dos agentes.
3. A **Estrutura de Informação de Gerenciamento** (ou *SMI-Struture of Management Information*) define as regras de como os **objetos gerenciáveis** são descritos e como ter acesso às operações do protocolo de suporte. O conjunto de definições das informações de gerenciamento sobre recursos é especificado na **Base de Informações de Gerenciamento - MIB** (*Management Information Base*) [07][18][20].
4. As **operações de gerenciamento**, implementadas pelo **protocolo** suporte, especificam as primitivas disponíveis para o usuário manipular as informações de gerenciamento.

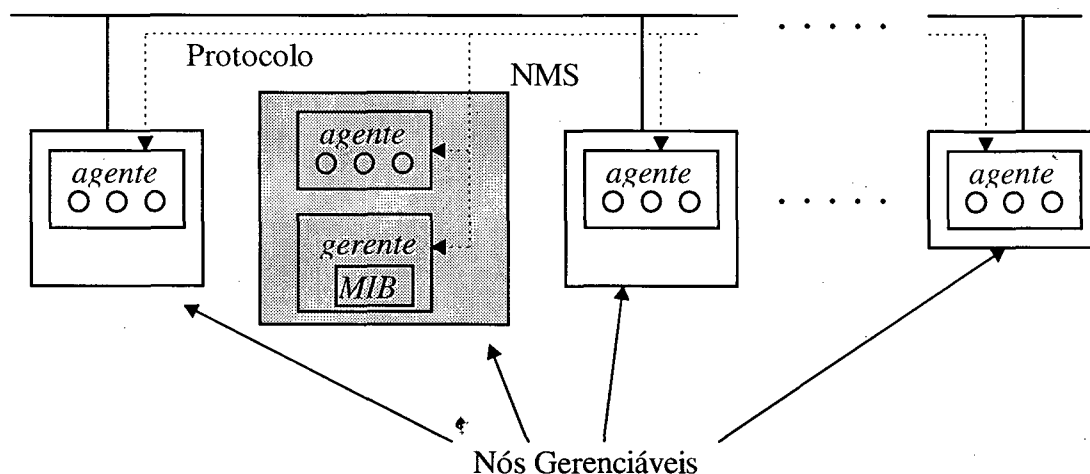


Figura 2-1 Modelo de Gerência

Dois padrões de gerência são mais relevantes no contexto atual: o **Padrão OSI** da ISO [30][34][39] e o **Padrão Internet** [19][20][07] que diferem basicamente na SMI empregada e nas operações do protocolo de gerenciamento. A estrutura de informações de gerenciamento do padrão Internet é mais simples, onde as próprias variáveis são consideradas como objetos, sem o suporte ao conceito de classes. Entretanto, no Padrão OSI, as informações estão encapsuladas em instâncias de classes de objetos. Portanto, costumeiramente é dito que o Padrão Internet é baseado em objetos e o Padrão OSI é orientado a objetos.

2.3 Representação de Dados

As aplicações de gerência, tanto agentes como gerentes, requerem que os dados recebidos ou enviados pela rede, de pedidos e respostas, estejam em um formato padrão, independentemente de onde tenham sido originados. Isto evita que cada aplicação tenha que se preocupar com detalhes próprios de cada máquina na rede.

Além disso, deve haver uma linguagem padrão para descrever os dados, de forma que seja abstrata em relação a representação interna de cada máquina. Portanto, a aplicação deve obter independência sintática da representação interna dos dados.

2.3.1 Sintaxe Abstrata

As informações intercambiáveis por aplicações de gerência são representadas por estruturas de dados. Estas estruturas de dados podem ser definidas pela composição de novos tipos ou serem de tipos primitivos. Por exemplo, a informação referente a um aluno pode ser representada por uma estrutura de dados “Aluno” que contém atributos como nome do aluno, número de matrícula e endereço. Atributos como o número de matrícula podem ser de um tipo primitivo, tal como um número inteiro. Entretanto, o atributo endereço requer uma outra estrutura composta, formada por rua, número, prédio e apartamento.

Uma sintaxe abstrata [39] para uma estrutura de dados é uma especificação da sua organização de forma independente da representação interna utilizada nas máquinas. A sintaxe abstrata torna possível a definição de novos tipos de dados e a atribuição de valores a estes tipos definidos.

A ISO definiu o ASN.1 (*Abstract Syntax Notation One*) como uma linguagem própria para a especificação de estruturas de dados de forma abstrata. A figura 2.2 mostra um exemplo da especificação em ASN.1 da estrutura “Aluno” como foi definida anteriormente.

ASN.1 é uma linguagem de descrição que possui duas características principais: uma notação usada em documentos facilmente interpretáveis e uma representação compacta da informação usada em protocolos de comunicação. Em ambas as características, o ASN.1 remove possíveis ambigüidades de representação e significado.

```
Aluno := [APPLICATION 0] IMPLICIT SET
{
    Nome          [1] VisibleString,
    Matricula     [2] IMPLICIT INTEGER,
    Endereço
}
Endereco = [APPLICATION 1] IMPLICIT SEQUENCE
{
    Rua           [1] VisibleString,
    Numero       [2] VisibleString,
    Predio       [2] VisibleString OPTIONAL,
    Apartamento  [3] VisibleString OPTIONAL
}
```

Figura 2-2 Especificação em ASN.1 da estrutura “Aluno”

2.3.2 Identificador de Objeto

Um identificador de objeto é uma seqüência de números inteiros não negativos representando os rótulos dos nós de uma árvore. A árvore consiste de um nó raiz conectada, através de arestas, a outros nós rotulados denominados de nós filhos. Cada nó filho por sua vez pode, ter arbitrariamente outros nós filhos, também rotulados, podendo este processo continuar até um nível arbitrário de profundidade. Além do número associado a um rótulo, cada nó pode ter também uma breve descrição textual.

Por exemplo, o descritor

1.3.6.1.2.1.4.3

identifica o objeto encontrado iniciando na raiz, movendo-se para o nó de rótulo filho 1, em seguida, movendo-se para o nó filho 3 e assim, sucessivamente até chegar ao último nó filho 3.

2.4 Axioma Fundamental do Gerenciamento de Rede Internet

O sucesso da Internet pode, em grande parte, ser creditado à simplicidade do protocolo IP e aos requisitos mínimos exigidos para implementá-lo.

Os tipos de equipamentos que podem assumir o papel de um nó gerenciável em uma rede é muito diversa. Entretanto, uma diretriz similar à definição do IP deve ser adotada para uma plataforma de gerenciamento [19]:

O impacto na adição de gerenciamento de redes ao nó gerenciável deve ser mínimo, refletindo o menor denominador comum.

Talvez esta preocupação seja exagerada quando se pensa em equipamentos de interconexão de redes com custos elevados e funcionalidade complexa, por exemplo, aqueles que atuam na camada 2 e 3 (como *brigdes*, roteadores e *switches*). Mas a adição de gerenciamento em dispositivos como *hubs* e repetidores eleva muito os custos, freqüentemente fazendo com que mais que dobre o preço do equipamento.

Desta forma, a implementação das funções de gerenciamento dos nós gerenciáveis deve ser a mais leve possível, o que implica em deslocar a complexidade da gerência para a Estação de Gerenciamento (NMS). Nota-se que na Internet, há muito mais nós gerenciáveis do que estações NMS, portanto, esta razão também torna-se uma questão econômica.

2.5 Estrutura da Informação de Gerenciamento

A Estrutura da Informação de Gerenciamento (*SMI - Structure of Management Information*) [21] define as regras para a descrição da informação de gerenciamento. O objetivo da SMI é permitir a descrição da informação de gerenciamento independente de qualquer detalhe de implementação.

2.5.1 MIB

Juntamente com a SMI, foi definida uma coleção de objetos gerenciáveis comuns, conhecida como Base de Informação de Gerenciamento (*MIB - Management Information Base*) [22]. A MIB pode ser visualizada como uma base de dados residindo em uma memória virtual. A SMI então, define o esquema desta base.

Depois do surgimento da primeira MIB, os trabalhos foram continuados até surgir uma nova MIB, denominada de MIB-II. Os trabalhos tinham entre outros objetivos, criar novos objetos gerenciáveis que refletissem os avanços tecnológicos e requisitos operacionais de novos equipamentos de rede, além de melhorar a clareza de certas especificações da MIB. A nova MIB-II, mais completa, manteve a compatibilidade com a MIB anterior.

2.5.2 SMI

A SMI especifica que todas as variáveis da MIB devem ser definidas e referenciadas usando a linguagem *Abstract Syntax Notation One* (ASN.1) da ISO. Além de manter documentos padrões sem ambigüidades, ASN.1 facilita a implementação de protocolos de gerenciamento de redes e garante a interoperabilidade.

2.5.3 Objetos Gerenciáveis

Cada objeto gerenciado é descrito usando-se a macro OBJECT TYPE em ASN.1 definida na figura 2.3.

As cláusulas da macro OBJECT-TYPE são SYNTAX, ACCESS e STATUS:

- **SYNTAX**

A sintaxe de um objeto gerenciável define o tipo de dados que o descreve. Na macro OBJECT-TYPE, a sintaxe é definida por um tipo de dados *ObjectSyntax*, definida mais adiante.

- **ACCESS**

Define o nível de acesso as instâncias do objeto gerenciável:

- *read-only* - podem ser lidas, mas não modificadas.
- *read-write* - podem ser lidas ou modificadas.

- **write-only** - podem ser modificadas, mas não lidas.
- **not-accessible** - não podem ser modificadas, nem lidas.

Entretanto, o perfil de gerenciamento usado pelo objeto gerenciável pode restringir os níveis de acesso definidos na macro.

```

OBJECT-TYPE MACRO :=
BEGIN
    TYPE NOTATION := "SYNTAX" type (TYPE ObjectSyntax)
                        "ACCESS" Access
                        "STATUS" Status
    VALUE NOTATION := value (VALUE ObjectName)

        Access := "read-only"
                    | "read-write"
                    | "write-only"
                    | "not-accessible"

        Status := "mandatory "
                    | "optional "
                    | "obsolete"
END

ObjectName := OBJECT IDENTIFIER
  
```

Figura 2-3 Definição da Macro OBJECT TYPE

- **STATUS**

Informa a obrigatoriedade da implementação da instância do objeto gerenciável pelo nó.

Pode ser um dos seguintes valores:

- **mandatory** - implementação obrigatória.
- **optional** - implementação não obrigatória.
- **obsolete** - os nós não devem mais implementar este objeto.

Por exemplo, o objeto *sysDescr* pode ser definido como:

```

sysDescr OBJECT-TYPE
    SYNTAX OCTET STRING
    ACCESS read-only
  
```

STATUS mandatory

:= { system 1 }

2.5.4 Identificação dos Objetos Gerenciáveis

Um identificador de objeto é um nome atribuído por uma autoridade e que faz parte de uma árvore de nomes, sendo um nó desta [40]. Os objetos gerenciáveis usados pela Internet estão na subárvore representada pela figura 2.4, definida pelo ramo:

1.3.6.1

e cujo identificador de objeto é

internet OBJECT IDENTIFIER := { iso org(3) dod(6) 1 }

2.5.5 Sintaxe

Na figura 2.3, a sintaxe dos objetos gerenciáveis é definida pelo tipo de objeto ASN.1 *ObjectSyntax*.

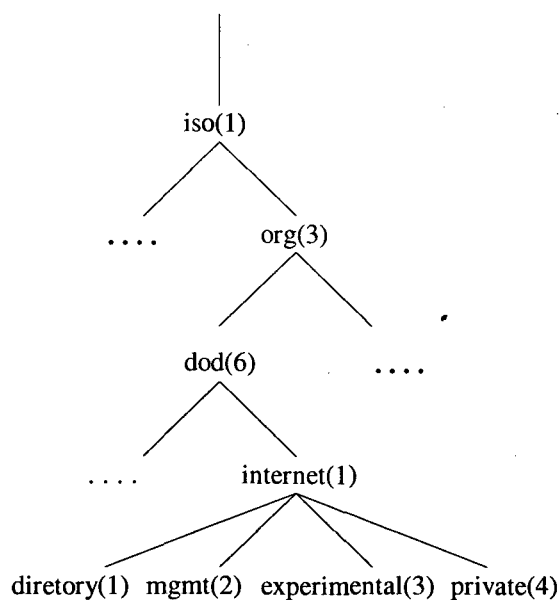


Figura 2-4 Subárvore da Internet

ObjectSyntax é definido como uma estrutura CHOICE, ou seja, o tipo é uma escolha entre os tipos definidos na estrutura, podendo ser:

- **Simples** - refere-se a um dos quatro tipos primitivos: INTEGER, OCTET STRING, OBJECT IDENTIFIER e NULL.
- **Aplicação** - refere-se a um dos tipos especiais definidos na SMI e que são mostrados na quadro 2.1.
- **Simples Construído** - refere-se a um dos dois tipos *list* ou *table*. A forma ASN.1 é representada na figura 2.5, sendo que os elementos em SEQUENCE não podem ser opcionais. *list* é usado para formar colunas em uma tabela e *table* é usado para representar tabelas.

```

<list> ::= SEQUENCE
{
    <type1>,
    .
    .
    <typeN>
}
<table> ::= SEQUENCE OF <list>

```

Figura 2-5 Definição de *list* e *table*

Tipo	Significado Representado	Definição ASN.1
IpAddress	endereço IP	IpAddress := [APPLICATION 0] IMPLICIT OCTET STRING(SIZE (4))
NetworkAddress	endereço em uma das diversas famílias de protocolos possíveis	NetworkAddress := CHOICE {internet IpAddress }
Counter	inteiro não negativo que cresce monotonicamente até um valor max e depois retorna a zero.	Counter := [APPLICATION 1] IMPLICIT INTEGER (0..4294967295)
Gauge	inteiro não negativo que cresce ou decresce até um valor max	Gauge := [APPLICATION 2] IMPLICIT INTEGER (0..4294967295)
TimeTicks	inteiro não negativo que mede o tempo em centésimos de segundo	TimeTicks := [APPLICATION 2] IMPLICIT INTEGER (0..4294967295)
Opaque	um tipo de dado arbitrário	Opaque := [APPLICATION 4] IMPLICIT OCTET STRING

Quadro 2-1 Tipos definidos pela SMI da Internet

2.6 Protocolo de Gerenciamento de Rede Internet

O protocolo de gerenciamento de rede da Internet, o *Simple Network Management Protocol - SNMP* [27], segue o paradigma da **depuração remota** [19], ou seja, cada nó gerenciável é visto como tendo diversas variáveis. A monitoração do nó pode ser realizada lendo-se as suas variáveis e o controle pode ser efetuado alterando-se os seus valores.

O SNMP é um protocolo assíncrono, com envio de pedidos e obtenção de respostas, sendo baseado num enfoque de *polling* direcionado a *trap*. Quando um evento ocorre, o nó gerenciável envia um *trap* muito simples para a estação de gerência. Desta forma, a estação de gerência avisada, fica responsável por tomar as medidas adequadas, entre elas, inicializar outras interações com o nó gerenciado para determinar a natureza e a extensão do problema.

Os requisitos de transporte do SNMP são modestos, necessitando somente de um serviço de transporte sem conexão, sendo geralmente usado o UDP. Estes requisitos mínimos, além de satisfazerem o Axioma Fundamental, deixam que as aplicações de gerência decidam qual o grau de confiabilidade desejável, o qual pode ser muito importante quando atuando numa situação de extrema gravidade.

2.6.1 Fluxo das Operações

Genericamente, uma interação SNMP consiste em um pedido, seguida de uma resposta. Esta interação é representada na figura 2.6.

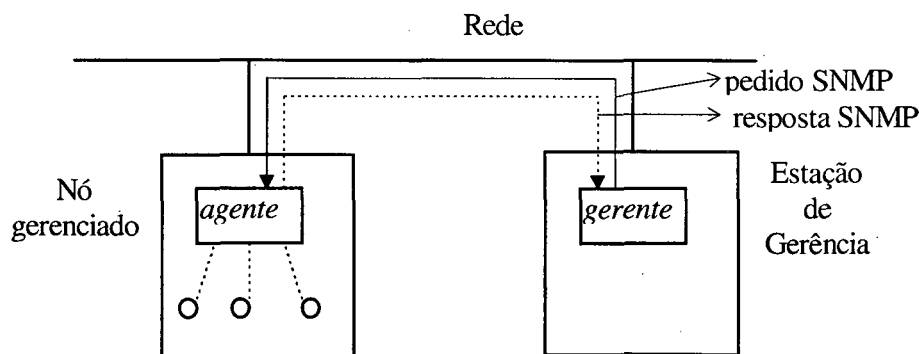


Figura 2-6 Representação da interação de um pedido SNMP

Há quatro resultados possíveis de uma operação SNMP:

1. Resposta sem exceção ou erro.
2. Resposta com uma ou mais exceções.
3. Resposta com um erro.
4. *Timeout*

Por outro lado, uma notificação de evento ou *trap*, gera uma simples PDU, a qual é repassada ao gerente. Esta interação é representada pela figura 2.7.

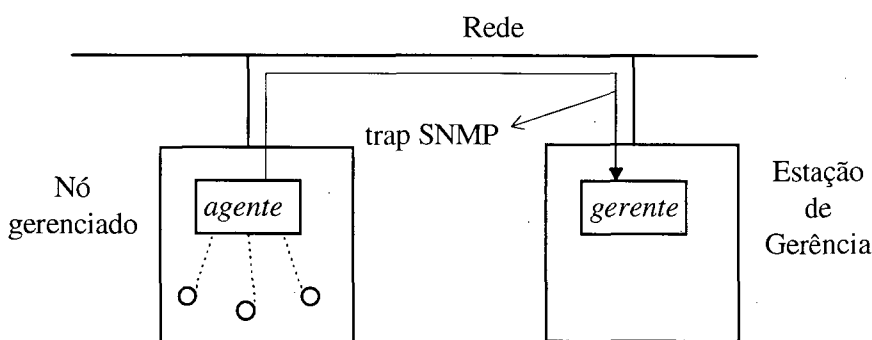


Figura 2-7 Representação da interação de um *trap* SNMP

2.6.2 Formato da Mensagem SNMP

Devido ao Axioma Fundamental, uma mensagem SNMP é muito simples e consiste de três partes principais, exibidas na figura 2.8:

1. Uma versão do protocolo que é não usada, pois diferentemente da maioria dos protocolos TCP/IP, não existe nenhuma negociação de versão no SNMP.
2. Um nome de comunidade usado para agrupar os nós gerenciáveis por um determinado gerente.
3. Uma área de dados que é dividida em unidades de dados de protocolos (PDUs).

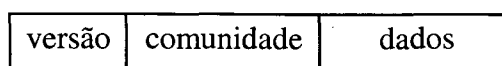


Figura 2-8 - Mensagem SNMP

2.6.3 Autenticação

O uso do nome da comunidade provê um esquema de autenticação, identificando como os dados são interpretados. A entidade de protocolo ao receber uma mensagem SNMP, passa três parâmetros para uma entidade de autenticação:

- O nome da comunidade.
- Os dados.
- O endereço de transporte da entidade que enviou a mensagem SNMP.

A entidade de autenticação, retorna uma das duas respostas abaixo, de acordo com o esquema de autenticação usado entre o grupo de nós gerenciáveis e o gerente:

- Uma **instância de dados** do tipo PDUs SNMP e a **identificação de autorização** da entidade SNMP enviando a mensagem. Desta forma, a entidade SNMP receptora pode processar o mensagem normalmente.
- Uma **exceção**, ou seja, houve uma falha na autenticação da mensagem e a entidade SNMP receptora pode, dependendo da configuração, gerar um trap *authenticationFailure*.

Até o momento, unicamente um esquema de autenticação denominado **trivial** é empregado, o qual apresenta várias falhas de segurança.

2.6.4 PDUs

Uma PDU SNMP pode consistir de uma das cinco PDUs usadas para os pedidos da maioria das operações normais e uma PDU de *trap*, usada para notificações. Os campos das PDUs para pedidos de operações normais são:

- **request-id** - É um inteiro usado pelo gerente para distinguir os pedidos feitos, permitindo a este, enviar um próximo pedido sem necessitar obter antes, a resposta do pedido anterior. As respostas que chegam podem então, serem relacionadas com os pedidos correspondentes feitos. Outra aplicação é evitar que possa haver mensagens e operações duplicadas na rede.

- **error-status** - Se o valor deste campo é diferente de zero, ele indica um exceção ocorrida no processamento do pedido.
- **-index** - Se diferente de zero, indica qual variável no pedido tem um erro. Este campo é diferente de zero unicamente para erros *noSuchName*, *badValue*, *readOnly*. Neste caso, é o deslocamento positivo dentro do campo *variable-bindings* (a primeira variável é dita ser de deslocamento 1).
- **variable-bindings** - Uma lista de variáveis, cada uma contendo um nome e um valor. A porção de valor de uma variável não é significativa para tipos de dados que são PDUs *GetRequest* e *GetNextRequest*. Por convenção o valor é sempre uma instância de tipo de dados NULL em ASN.1. Contudo, a entidade de recepção SNMP deverá simplesmente ignorar o valor que é fornecido pela entidade enviando o pedido SNMP.

Os campos de uma PDU de *Trap* são:

- **enterprise** - É o valor do *sysObjectID* do agente.
- **agent-addr** - É o valor do *NetworkAddress* do agente.
- **generic-trap** - Indica um evento genérico tal como a reinicialização do agente, queda das interfaces de rede ou falha na autenticação.
- **specific-trap** - O valor de *specific-trap* se diferente de zero, identifica qual o trap *enterpriseSpecific* que ocorreu.
- **time-stamp** - É o valor do objeto *sysUpTime* do agente quando o evento ocorreu.
- **variable-bindings** - Indica uma lista de variáveis contendo informação sobre *otrap*.

2.6.5 Operações SNMP

Depois de realizar o passo de autenticação e obtenção do perfil da comunidade associada, uma entidade SNMP pode realizar as seguintes operações:

1. Operações de Recuperação: **Get e Get-Next**
2. Operação de modificação: **Set**
3. Operação de Notificação: **Trap**

2.6.6 Recuperação de Valores de Objetos

Cada instância tem um identificador de objeto que é formado concatenando-se o nome do objeto com um sufixo. Desta forma, um ordenamento lexicográfico pode ser estabelecido sobre todas as instâncias, extremamente útil nas operações *Get-request* e *Get-Next-request*.

- ***Get-request***

Se o gerente conhece precisamente as instâncias da informação de gerenciamento desejadas, então ele insere uma operação *Get-request*.

Para cada variável na lista de variáveis do pedido, a instância nomeada é restaurada no contexto do perfil da comunidade. Se uma instância não existe, uma *Get-response* é retornada com o erro *noSuchName*. De outra forma, uma *Get-response* é retornada idêntica ao pedido, contendo a lista de variáveis com os respectivos valores de cada instância.

Nota-se que desta forma, não é possível com um código de erro *noSuchName*, saber se um objeto não é implementado pelo agente ou simplesmente não tem a instância desejada.

- ***Get-Next-request***

Se o gerente conhece qual o ramo da árvore da MIB onde está a variável, mas não especificamente, o nome da variável, então ele pode inserir uma operação *Get-Next-request*.

Para cada variável na lista de variáveis do pedido, é restaurada na ordem lexicográfica, a próxima instância nomeada no contexto do perfil da comunidade. Uma *Get-response* é então, retornada idêntica ao pedido, exceto que o nome e os valores correspondentes das variáveis na lista do pedido são devidamente preenchidas. Se é alcançado o final da ordem lexicográfica, uma *Get-response* é retornada com o erro *noSuchName*.

A operação *Get-Next* usa intensamente, e bem, a ordem lexicográfica dos nomes das variáveis. Por exemplo, a operação

get-next({sysDescr, unSpecified})

retorna o nome e o valor da próxima instância na visão da MIB, que deverá ser

sysDescr.0

Desta forma, pode-se usar a operação *Get-Next* para verificar se um objeto é suportado por um determinado agente, sem especificar qualquer instância.

2.6.7 Modificação de Valores de Objetos

A operação *Set-request* está disponível para pedidos de modificações em objetos. Para cada variável no pedido, a instância nomeada é identificada no contexto do perfil da comunidade.

Se a instância não existe, uma *Get-response* é retornada com erro *noSuchName*.

Se a instância existe, mas não são permitidas operações de escrita, uma *Get-response* é retornada com erro *readOnly*. Se uma instância existe e são permitidas operações de escrita, mas o valor informado tem um erro no seu formato (erro sintático) ou não está dentro do intervalo válido, uma *Get-response* é retornada com o valor *badValue*.

2.6.8 Notificação de Eventos

Quando um agente necessita enviar uma notificação de um evento extraordinário, uma operação *Trap* é gerada, podendo ser enviada para um ou mais gerentes com uma comunidade apropriada.

Na recepção de uma PDU *Trap*, o gerente então, toma as ações apropriadas em resposta ao evento.

2.6.9 Recepção de uma Resposta

A PDU *Get-response* é usada para enviar uma resposta para um gerente.

O gerente verifica, na lista de pedidos anteriormente enviados, os seus *request-ids* para localizar de qual pedido corresponde a resposta. Se não encontra, a resposta é desconsiderada e removida. Se encontra, o gerente então processa a resposta.

2.7 SMUX

O código de um agente SNMP geralmente envolve a manipulação de estruturas complexas do núcleo do sistema operacional, podendo fazer com que a adição de novos agentes por parte do usuário seja muito difícil, mesmo que ele não necessite utilizar tais estruturas.

Em vista disso, muitas implementações fornecem um protocolo que permite ao usuário escrever processos bem menos complexos, os quais são chamados **subagentes**. Um subagente é um processo que comunica-se com o agente SNMP local e controla determinados objetos definidos em uma MIB privada, ou seja, que não são definidos no conjunto padronizado dos objetos da MIB (MIB-I e MIB-II) da *Internet*.

Quando um subagente é criado, ele precisa comunicar-se com um agente SNMP e, uma das maneiras de estabelecer esta comunicação é através do protocolo SMUX [23] [41]. O subagente e o agente formam desta forma, um **par SMUX**.

Quando um subagente, condicionado ao SMUX, deseja exportar um módulo de sua MIB, ele inicia uma conexão SMUX e solicita o registro da subárvore privada ao agente. O agente atende as operações de gerenciamento enviadas pelo gerente, redirecionando ao subagente aquelas referentes aos objetos de sua subárvore registrada. O subagente pode ainda, enviar ao agente notificações à cerca de situações anormais ocorridas em determinados objetos. O agente então, pode redirecionar estas notificações ao gerente.

A figura 2.9 ilustra a comunicação entre agente SNMP e o processo usuário através do protocolo SMUX:

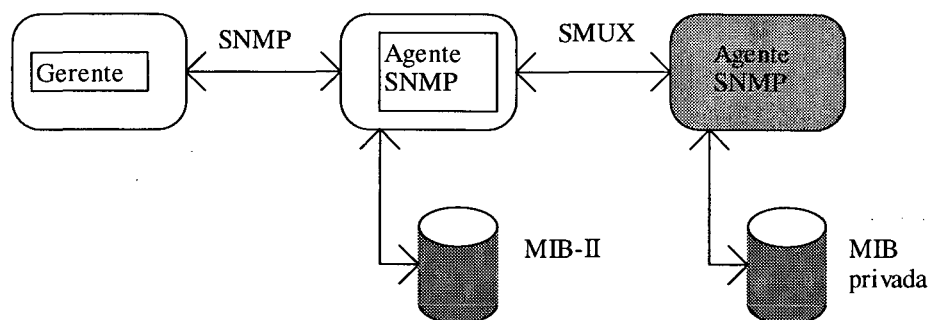


Figura 2-9 Protocolo Multiplexador do SNMP - SMUX

2.7.1 Descrição do Protocolo SMUX

O protocolo SMUX é muito simples. Seu princípio de funcionamento baseia-se no modelo cliente-servidor, onde o agente SNMP, um dos processos do par SMUX, atua como servidor e o programa que implementa os objetos gerenciáveis, atua como cliente, que é o outro processo do par SMUX.

- **Fase de Estabelecimento de Associação**

O agente SNMP fica à espera da chegada de pedidos de conexões como mostra a figura 2.10. Quando um subagente deseja conectar-se com o agente, uma associação passa a ser inicializada pelo subagente, que insere uma primitiva *OpenPDU*. A associação pode ser rejeitada pelo agente através da inserção da primitiva *ClosePDU*, fechando a conexão. Do contrário, se o agente não inserir nenhuma resposta, então a associação é aceita.

Tanto o subagente, quanto o agente podem terminar, em qualquer momento, a associação inserindo a *ClosePDU*.

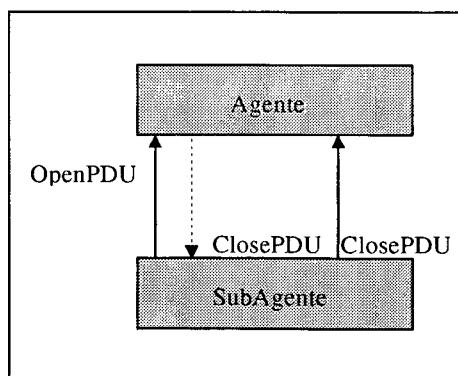


Figura 2-10 Estabelecimento e encerramento de uma associação SMUX

- **Fase de Comunicação SNMP**

Para cada subárvore definida num módulo da MIB que o subagente deseja registrar, ele insere a primitiva *RReqPDU*, como mostra a figura 2.11. O agente, assim que recebe a primitiva, processa o pedido e responde com a primitiva *RRspPDU*.

O agente SNMP pode receber uma PDU SNMP *Get-request*, ou a *Get-Next-request*, ou ainda, a *Set-request* de um gerente. O agente envia uma PDU SMUX contendo o pedido SNMP para o subagente que registrou, com maior prioridade, a subárvore com as variáveis do pedido. Assim que o subagente recebe a PDU, ele realiza a operação pedida e insere uma *PDU Get-response* correspondente ao pedido. O agente verifica esta resposta e reenvia a PDU para o gerente.

Da mesma forma, quando o subagente deseja inserir um *trap*, ele insere uma *PDU Trap* para o agente. O agente recebe a PDU e a redireciona para o gerente.

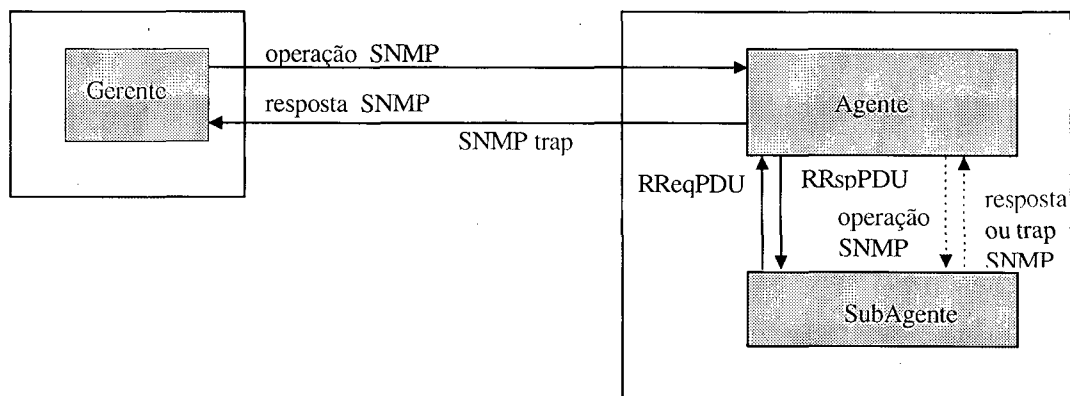


Figura 2-11 Comunicação SNMP de um subagente

2.7.2 Registro da subárvore da MIB

Um número inteiro, no intervalo de 0 a $2^{31}-1$, é associado com cada registro. Este número significa a prioridade do registro, sendo que, quanto menor o valor, maior a prioridade. Desta forma, pode-se dizer que uma subárvore e uma prioridade identificam unicamente um par SMUX.

Vários subagentes podem tentar registrar a mesma subárvore, mas eles devem utilizar prioridades diferentes. Se um subagente tentar registrar uma subárvore com uma prioridade já utilizada, o agente repetitivamente incrementa a prioridade até um valor não usado. O valor especial -1 informa ao agente para escolher a prioridade mais alta disponível, baseada na sua configuração local.

O subagente com a prioridade mais alta é consultado para todas as operações em uma subárvore. Neste sentido, pode ocorrer o efeito de montagem de subárvore. Quando um

subagente registra uma subárvore que contém outra já registrada por outro subagente, toda a consulta para esta subárvore irá para aquele que registrou a subárvore maior. Por exemplo, se um subagente registrou a variável *sysDescr* e outro registra mais tarde a subárvore *system* a qual inclui *sysDescr*, todos os pedidos para *sysDescr* serão enviados para o último subagente.

2.7.3 Remoção do Registro

Um subagente pode remover os registros de subárvores. Se a prioridade dada na RReqPDU é -1, então a registro de prioridade mais elevada é selecionada para deleção. Se a prioridade é diferente de -1, então unicamente o registro com prioridade igual é selecionado para deleção.

2.7.4 Variáveis nos Pedidos

Uma PDU SNMP pode conter um ou mais variáveis, mas o agente geralmente processa as operações para cada variável seqüencialmente. Para isto, antes de processar cada variável, o agente pode bloqueá-la, não permitindo a alteração de seu valor por outro pedido enquanto está realizando o pedido que causou o bloqueio.

3. REDES NEURAIS ARTIFICIAIS

Tradicionalmente, as soluções algorítmicas são o enfoque adotado para a solução dos problemas resolvidos em computador, mesmo quando refere-se a Inteligência Artificial. Entretanto, este enfoque origina algoritmos que analisam uma grande quantidade de dados e são testados num contexto restrito. Quando são aplicados a problemas reais, a capacidade de processamento e a memória dos computadores se mostram muitas vezes insuficientes.

Entretanto, para um ser humano, este tipo de problema é extremamente fácil de ser solucionado. A experiência humana está organizada em uma rede de células neurais. Os métodos de “cálculo” ou “dedução” são totalmente diferentes dos métodos algorítmicos ou sistemas de regras.

3.1 Soluções Algorítmicas

A Inteligência Artificial procura solucionar problemas facilmente resolvidos pelo cérebro humano, mas de solução algorítmica complicada. Tome-se como exemplo o jogo de xadrez, onde um lance pode ter uma enorme quantidade de possibilidades. O cálculo algorítmico de qual a melhor solução pode consumir muitos recursos da máquina como tempo de ocupação da CPU e quantidade de memória RAM utilizada.

Outro exemplo são os Sistemas Especialistas, muito utilizados nos últimos anos. Estes sistemas usam o enfoque algorítmico para capturar a experiência de um especialista humano usando dois componentes [01]:

1. Uma lista regras.

2. Uma maneira de inferir e tirar conclusões sobre cada regra e alguns fatos fornecidos para responder uma questão.

Por exemplo, tendo-se a seguinte regra:

Se X é filho de A e Y é filho de A, então X e Y são irmãos

e os seguintes fatos:

Maria é filha de Joana e Fred é filho de Joana

então usando a regra acima, poderia-se concluir que Maria e Fred são irmãos.

Contudo, nem sempre um enfoque algorítmico se mostra adequado. Tome-se o caso de reconhecimento da escrita humana. Algorítmicamente, a solução para reconhecimento de um texto (sem entendimento semântico ou sintático), passaria pelo reconhecimento de cada letra. Mas isto obrigaria que fossem armazenadas em uma base de dados, todas as possíveis formas que uma determinada pessoa possa escrever cada caracter, por exemplo, a letra A. Obviamente isto é impossível, pois nada garante que no texto a ser reconhecido, tenha somente formas de caracteres armazenados na base, ou seja, a pessoa que escreveu pode ter desenhado um caracter com uma ligeira modificação de uma forma inexistente na base de dados.

3.2 Características das Redes Neurais

As redes neurais artificiais, ou simplesmente, redes neurais, são inspiradas no cérebro humano. Elas são compostas por elementos que realizam funções elementares análogas aos neurônios, arranjados em uma estrutura que pode procurar imitar ou não o cérebro. Desta forma, as redes neurais apresentam algumas semelhanças marcantes com o cérebro humano, dentre as quais [26][42]:

- **Aprendizagem**

As redes neurais podem modificar seu comportamento em respostas a eventos ou fatos que ocorrem no ambiente externo. Estes eventos ou fatos fornecem um conjunto de entradas, as

quais podem ser acompanhadas de um conjunto de saídas desejado. Este conjunto, através de um algoritmo de treinamento, provoca um auto-ajuste na rede neural para produzir um conjunto de respostas, adequado e consistente com os padrões de entrada.

- **Generalização**

Depois de treinada, a resposta da rede pode ser insensível a pequenas variações na sua entrada. Esta característica é fornecida pela estrutura da rede e não devido a algum algoritmo intrínseco.

A generalização é útil para tratar com pequenas variações de um padrão, ou seja, facilita o tratamento de problemas relacionados a um mundo imperfeito.

3.3 Neurônio Artificial

O neurônio artificial tenta imitar o neurônio biológico. A figura 3.1 mostra um modelo que descreve a idéia do neurônio artificial [42].

Como pode ser visto, um conjunto de entradas x_1, x_2, \dots, x_n é aplicado à entrada do neurônio, formando um vetor X . Estas entradas são originadas das saídas de outros neurônios. Cada entrada no neurônio, é então, multiplicada por um peso correspondente, representando a sinapse do neurônio biológico. O conjunto dos pesos w_1, w_2, \dots, w_n forma o vetor W . Todas as entradas, multiplicadas pelo seu peso correspondente, são somadas algebricamente no bloco Σ , obtendo-se o valor NET. Portanto,

$$NET = x_1 w_1 + x_2 w_2 + \dots + x_n w_n \quad (1)$$

ou

$$NET = X.W \quad (2)$$

O bloco de soma Σ representa o corpo de um neurônio biológico.

A saída NET é em seguida processada por uma função F , denominada de **Função de Ativação**, produzindo a saída OUT. Esta função pode ser uma função linear simples como:

$$\text{OUT} = K(\text{NET}), \quad \text{onde } K \text{ é uma constante} \quad (3)$$

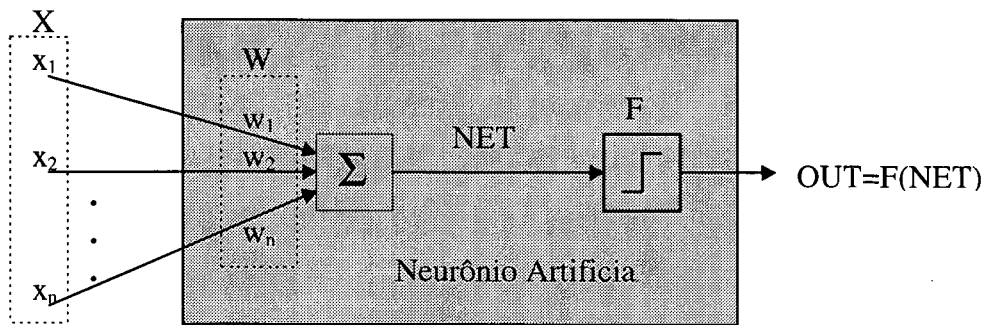


Figura 3-1 Neurônio Artificial

3.4 Função de Ativação

A função de ativação F pode ser uma função simples ou uma mais complexa. Exemplos de funções simples podem ser:

1. $F = K(\text{NET})$ onde K é uma constante
2. F é uma função Limiar:

$$F = 1 \text{ se } \text{NET} > L, \text{ onde } L \text{ determina um valor mínimo.}$$

$$F = 0 \text{ caso contrário}$$

Se F comprime o intervalo de valores de NET , não permitindo que ultrapasse certos valores (pequenos), F é chamada de função *squashing*. Esta característica resolve o problema do dilema de saturação de ruído [13] [15], permitindo a rede trabalhar com valores altos e pequenos ao mesmo tempo.

Exemplos de funções de ativação muito usadas são as funções Sigmóide

$$\left(F(x) = \frac{1}{1 + e^{-x}} \right) \text{ e a Tangente-hiperbólica } (F(x) = \tanh(x)).$$

3.5 Redes Neurais Artificiais

Um único neurônio pode realizar certos reconhecimentos de padrões simples. Entretanto, unido-se vários neurônios podemos formar uma rede neural que consegue reconhecer uma quantidade maior e mais complexa de padrões. De fato, o poder computacional da rede neural advém das conexões entre os neurônios, ou seja, das sinapses formadas.

3.5.1 Redes Neurais de uma Única Camada

Uma rede neural de uma camada é uma rede simples que une vários neurônios como mostrado na figura 3.2. Nesta figura são mostradas dois grupos de neurônios: o grupo mais a esquerda e o grupo mais a direita.

O grupo mais a esquerda unicamente serve para distribuir as entradas, não realizando nenhum cálculo. Portanto, não é considerada como uma camada propriamente dita, não contando como uma camada de rede neural.

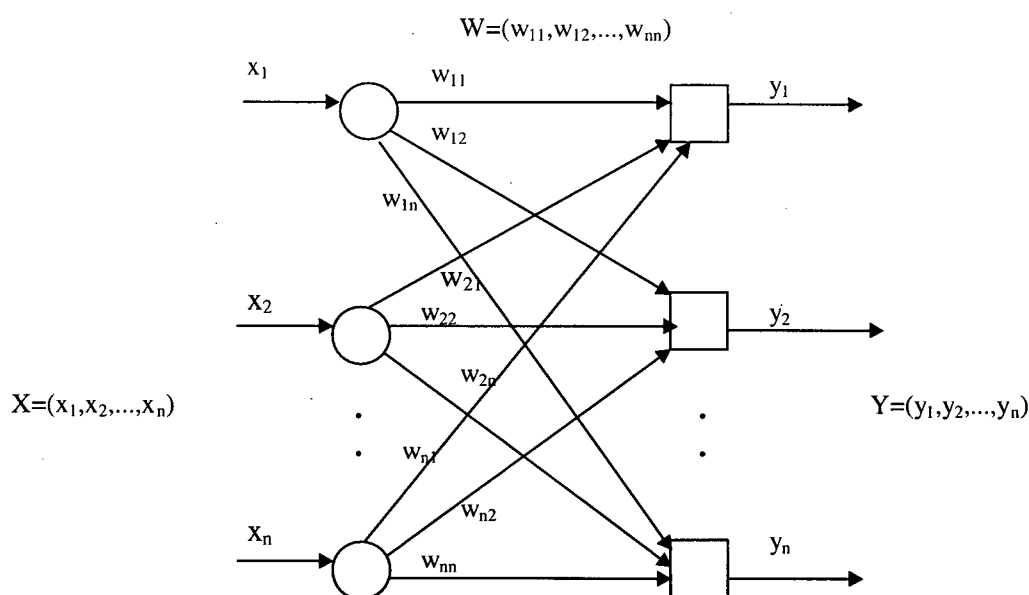


Figura 3-2 Rede Neural de Uma Camada

O grupo mais a direita é composto por neurônios que realizam a atividade neural, formando a camada neural propriamente dita. Por esta razão, são representados com um

quadrado. Cada elemento x_i de X , é apresentado aos neurônios desta camada através de conexões com pesos w_{ij} . Cada neurônio então, soma as entradas e coloca o resultado na saída.

Cada peso w_{ij} é um elemento da matriz W , onde i é o número da entrada e j o número do neurônio correspondente. Portanto, NET é calculado através da multiplicação do vetor X pela matriz W :

$$NET = X.W \quad (4)$$

As redes neurais podem ter muitas das suas conexões removidas durante a atividade normal da rede. Isto pode ser representado atribuindo-se aos pesos correspondentes, o valor zero. Conexões entre os neurônios de entrada e os de saída podem existir, mas esta configuração não será tratada aqui.

3.5.2 Redes Neurais de Múltiplas Camadas

Estudos neurológicos deduzem que o cérebro humano é composto por camadas de neurônios, portanto, as redes neurais de múltiplas camadas modelam melhor a estrutura cerebral [13] [29]. Além disso, tem-se provado que redes neurais de múltiplas camadas tem maiores capacidades computacionais do que redes de camada simples.

A figura 3.3 mostra uma rede de múltiplas camadas (2 camadas). Como pode ser visto, as redes neurais de múltiplas camadas são realizadas agrupando-se em cascata um grupo de redes de camadas simples. A saída de uma camada provê a entrada para outra.

3.6 Treinamento em Redes Neurais

A capacidade de aprendizagem através do treinamento apresentada pelas redes neurais é a sua característica mais marcante. Portanto, um paralelo entre o cérebro humano e as redes neurais artificiais pode ser feito, onde o desenvolvimento intelectual também passa por uma fase de treinamento.

Entretanto, esta comparação é ainda muito limitada, pois as redes neurais estão apenas começando a serem estudadas e muitos problemas encontrados precisam ser melhor compreendidos.

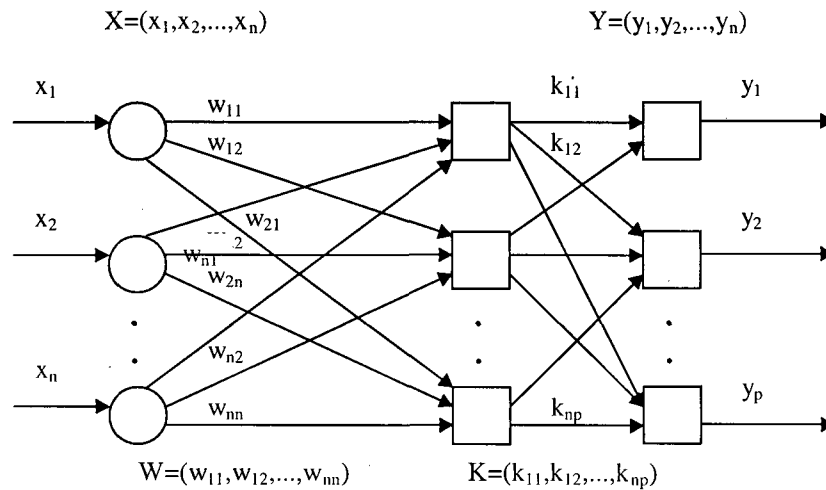


Figura 3-3 Rede Neural de Múltiplas Camadas

3.6.1 Fase de Treinamento

Uma rede neural está treinada quando, aplicando-se um conjunto ou vetor de entradas na rede, se obtém um conjunto ou vetor de saídas desejadas. Neste caso, o arranjo dos valores dos pesos da rede está com os valores adequados para o contexto do problema em que a rede foi treinada. Pode-se dizer que os pesos representam as sinapses de uma rede neural biológica.

Conseqüentemente, deve haver uma fase de treinamento para ajustar os pesos da rede. Nesta fase de treinamento, o vetor de entrada é aplicado seqüencialmente à rede, enquanto se ajusta, de acordo com um algoritmo de treinamento, os pesos para se obter o vetor de saída desejado. Estes algoritmos de treinamento podem ser **supervisionados** ou **não-supervisionados** [31].

3.6.2 Treinamento Supervisionado

No treinamento supervisionado, para cada vetor de entrada apresentado à rede, há um vetor representando a saída desejada, ambos formando um par de treinamento.

Para cada vetor de entrada aplicado à rede, uma saída é obtida. Esta saída é então comparada com a saída desejada correspondente. A diferença (ou erro) é realimentada na entrada, fazendo com que os pesos sejam ajustados, de acordo com um algoritmo de treinamento, visando minimizar o erro provocado.

Desta forma, todos os vetores do conjunto de treinamento são aplicados seqüencialmente e erros são calculados para cada vetor, sendo que cada erro provoca um ajuste nos pesos da rede neural. Esta seqüência de treinamento continua até que se atinja um erro, dentro de uma tolerância aceitável, para o conjunto inteiro de treinamento.

3.6.3 Treinamento Não-Supervisionado

O treinamento não-supervisionado não necessita de vetores de saída desejada e portanto, nenhuma comparação é realizada entre a saída obtida e um saída desejada. O conjunto de treinamento consiste inteiramente nos vetores de entradas.

O algoritmo de treinamento ajusta os pesos da rede de forma que produzam vetores de saída consistentes. A consistência dos valores significa por exemplo, que a aplicação de dois vetores de treinamento similares produzem as mesmas saídas. Portanto, o algoritmo de treinamento extrai as propriedades estatísticas do conjunto de treinamento e agrupa os vetores similares em classes específicas.

3.7 Realimentação de Redes Neurais

Outra característica de uma rede neural é a ausência ou não de realimentação, ou seja, conexões da saída de uma camada para entrada da mesma camada ou de camadas anteriores. As redes que apresentam realimentação são ditas **redes *feedbacks*** e aquelas que não apresentam, são ditas **redes *feedforward*** [01] [17][29].

3.7.1 Redes Feedforward

Redes *feedforward* não tem memória. Sua saída é inteiramente determinada pelas entradas correntes e os valores dos pesos correspondentes.

As redes *feedforward* de única camada, como esquematizado na figura 3.4, não apresentam problemas no treinamento. Sua saída é determinada assim que a entrada é conhecida. As redes *feedforward* de múltiplas camadas tem neurônios os quais não estão diretamente conectados à saída, denominados de **neurônios ocultos**. A informação de treinamento consiste unicamente de padrões de entrada e as saídas correspondentes, não contendo uma especificação de qual deverá ser o ajuste das entradas nas camadas ocultas.

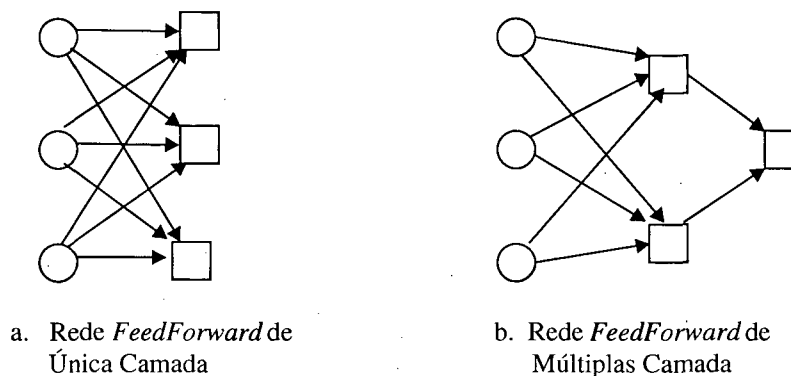
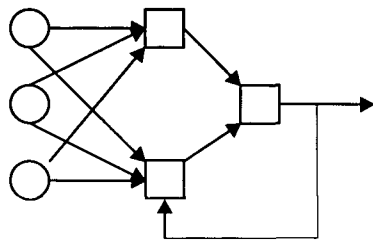
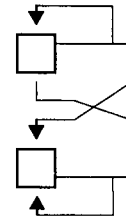


Figura 3-4 Redes Feedforward

3.7.2 Redes Feedback

As redes *feedback* realimentam o sinal de entrada com o sinal de saída, portanto, a saída da rede é determinada pela entrada corrente e sua saída anterior.

A figura 3.5 mostra duas redes *feedback*. Em 3.5.a, a rede tem somente um laço de realimentação, enquanto a rede em 3.5.b tem dois laços. A rede em b é dita ser inteiramente autoassociativa, porque a saída desejada, realimentada na entrada, serve como padrão de treinamento. A rede em a, é dita parcialmente autoassociativa, pois a saída desejada, é também realimentada na entrada, mas mesmo assim, a rede é alimentada com um padrão de treinamento.

a. Rede *feedback* parcialmente autoassociativab. Rede *feedback* inteiramente autoassociativa**Figura 3-5 Redes *FeedBack***

3.8 *Backpropagation*

O algoritmo *Backpropagation* é um método sistemático para treinamento de redes neurais de múltiplas camadas[43]. Ele tem uma forte fundamentação matemática e tem aberto muitos campos de aplicações para as redes neurais.

Neste texto, será usada a notação de Wasserman [42] por ser a mais apropriada aos objetivos do trabalho, desconsiderando alguma falta de rigor matemático.

3.8.1 O Neurônio Fundamental

A figura 3.6 exibe o neurônio usado como bloco fundamental de construção das redes neurais com algoritmo de treinamento *Backpropagation*.

Nestas redes, uma entrada o_i pode ser uma entrada externa ou a saída de um neurônio da camada anterior. Cada entrada o_i é então multiplicada pelo peso correspondente w_i e os resultados são somados no bloco Σ , resultando no valor NET.

Portanto, NET tem o valor:

$$\text{NET} = o_1w_1 + o_2w_2 + \dots + o_nw_n = \sum_{i=1}^n o_iw_i \quad (5)$$

Em seguida, NET passa por uma função de ativação F , produzindo o valor OUT:

$$\text{OUT} = F(\text{NET}) \quad (6)$$

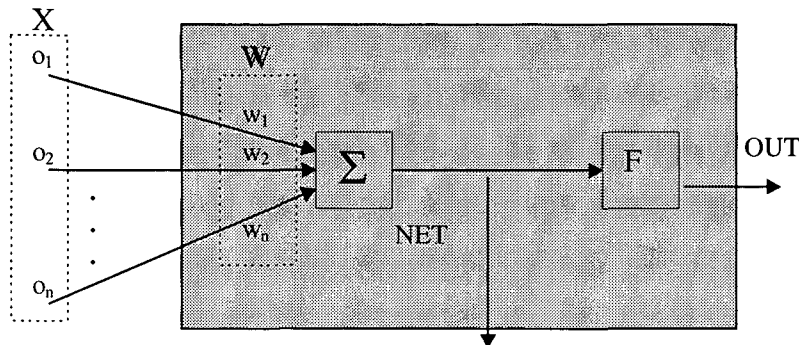


Figura 3-6 Neurônio Fundamental do Backpropagation

Há muitas funções de ativação F possíveis. O algoritmo *Backpropagation* requer que F seja diferenciável. Além disso, tendo em vista o problema da separabilidade linear [31][32][42], a função requerida deve ser não-linear. Exemplos de funções que observam estes requisitos, são as funções Sigmóide e a Tangente-hiperbólica [42]. A figura 3.7 mostra uma rede multicamadas que pode ser treinada pelo algoritmo do *Backpropagation*.

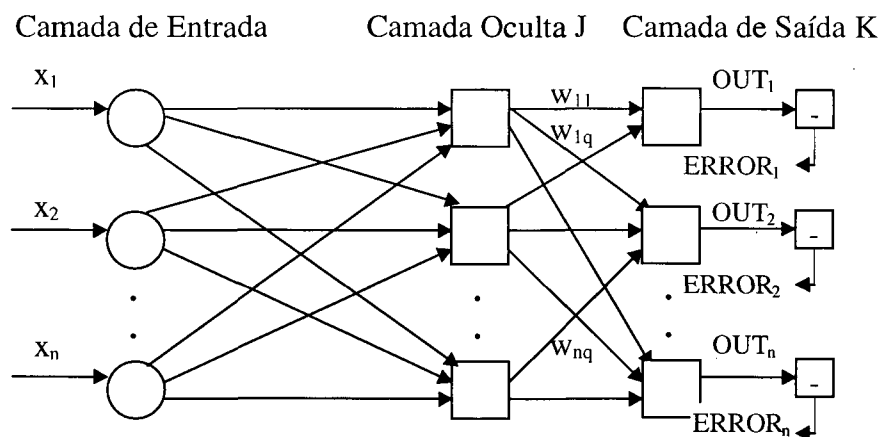


Figura 3-7 Redes Neurais de Múltiplas Camadas Treináveis pelo *Backpropagation*

3.8.2 O Algoritmo de Treinamento *Backpropagation*

Cada vetor de entrada tem um vetor de saída desejado correspondente, ambos formando um **par de treinamento**. O conjunto de treinamento é composto por todos os pares de treinamento usados para a rede neural.

Um requisito do algoritmo de treinamento *Backpropagation* é que os valores iniciais atribuídos aos pesos sejam randômicos [13]. Portanto, satisfeito este requisito inicial, os passos do algoritmo são os seguintes[42]:

Repita:

Para cada par de vetores no conjunto de treinamento faça :

1. Aplique à entrada da rede neural, o próximo vetor de entrada.
2. Calcule a saída obtida da rede neural.
3. Calcule o erro entre a saída obtida na rede neural e a saída desejada.
4. Ajuste os pesos da rede de maneira que minimize o erro obtido.

Fim para

Até que o erro obtido esteja dentro da tolerância desejada.

Nos passos 1 e 2, o sinal propaga-se na rede da entrada para a saída e portanto, constituem um passo *forward*. Enquanto isso, os passos 3 e 4 constituem um passo *backward*.

• **Passo Forward**

O cálculo em redes de múltiplas camadas é feito camada a camada, à partir da camada mais próxima até a camada de entrada. Na primeira camada, o valor NET de cada neurônio é calculado como a soma ponderada das entradas nos neurônios. Então, para cada neurônio, o valor NET é aplicado na função de ativação F, produzindo o valor OUT do neurônio correspondente na camada 1.

Então, o valor OUT de cada neurônio da camada 1, passa agora, a ser a entrada para os neurônios da 2 camada e o cálculo se repete. Este procedimento é repetido, camada à camada, até que o conjunto final de saídas na última camada seja produzido.

Usando-se uma notação vetorial, tem-se:

O representa o vetor de saída: $O = (o_1, o_2, \dots, o_n)$,

W representa o vetor de pesos: $W = (w_1, w_2, \dots, w_n)$ e

X representa o vetor de entrada: $X = (x_1, x_2, \dots, x_n)$

Então, para uma determinada camada, o valor O é dado por :

$$O = F(X.W)$$

Portanto, o cálculo da saída da última camada requer a aplicação da equação acima para cada camada da rede neural de modo que, o vetor de saída O seja o vetor de entrada X da próxima camada.

- **Passo *Backward***

O passo *backward* serve para o ajuste dos pesos das camadas. Para a última camada, um vetor de saída desejado é disponível e portanto, pode-se realizar um ajuste baseado na **Regra Delta** [10][28] [42]. Entretanto, nas camadas ocultas, não existe um vetor de saída desejada correspondente para cada camada, o que irá exigir o uso de um artifício para que a Regra Delta possa ser aplicada.

- **Ajuste de pesos da camada de saída**

A figura 3.8 mostra o ajuste de um peso de um neurônio p na camada oculta j , para o neurônio q na camada de saída k . Um sinal representando o erro na camada k é obtido a partir do resultado da subtração da saída obtida com a saída desejada. O sinal δ é então obtido pela multiplicação do erro obtido pela derivada F' , da função de ativação F : $OUT(1-OUT)$.

$$\delta = OUT(1-OUT)(TARGET-OUT) \quad (8)$$

Em seguida, δ é multiplicado pelo valor OUT de um neurônio da camada j . O resultado é por sua vez, multiplicado pela taxa de aprendizagem η e o resultado é adicionado ao peso. Portanto, temos as seguintes equações:

$$\Delta w_{pq,k} = \eta \delta_{q,k} OUT_{qj} \quad (9)$$

$$w_{pq,k}[n+1] = w_{pq,k}[n] + \Delta w_{pq,k} \quad (10)$$

onde:

$w_{pq,k}[n]$: valor de um peso do neurônio p na camada oculta, para o neurônio q na camada de saída no passo n .

$w_{pq,k}[n+1]$: idem para o mesmo neurônio, no passo $n+1$.

$\delta_{q,k}$: valor do peso no passo $n+1$.

OUT_{pj} : valor da saída (OUT) para o neurônio p na camada oculta j .

Por fim, para cada neurônio da camada de saída, um processo idêntico é realizado.

- **Ajuste de pesos para as camadas ocultas**

O algoritmo *Backpropagation* treina as camadas ocultas, propagando os erros obtidos na saída, de volta para a rede, camada à camada, ajustando os pesos correspondentes em cada uma.

Considere um neurônio na última camada oculta. No passo *forward*, através da interconexão dos pesos, o neurônio propaga sua saída para os neurônios na camada oculta. No treinamento, estes pesos agem no sentido inverso, ou seja, passam o valor δ da camada de saída para a camada oculta. Cada um desses valores é multiplicado pelo valor δ de cada neurônio que ele conecta na camada de saída. O valor δ necessário para o neurônio da camada oculta é produzido pela soma de todos os produtos e multiplicado pela derivada da função F .

$$\delta_{pj} = OUT_{pj}(1 - OUT_{pj})\left(\sum_q \delta_{q,k} w_{pq,k}\right) \quad (11)$$

Os pesos que entram na última camada oculta podem ser ajustados usando as equações do passo *forward*.

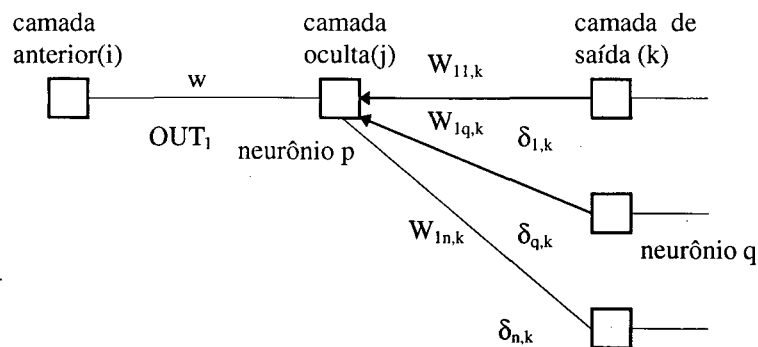


Figura 3-8 Ajustes dos Pesos

Portanto, para cada camada oculta, deve ser calculado o valor dos δ s correspondentes e todos os pesos associados com aquela camada devem ser ajustados. Isto deve ser repetido para todas as camadas retrocedendo até a camada de entrada.

Pode-se obter uma versão em notação vetorial [42] da equação acima. Considere o que δ s na camada de saída compõem o vetor D_k e o conjunto de pesos na camada de saída compõem a matriz W_k . Para calcular D_j , o vetor de δ s da última camada oculta, dois passos devem ser seguidos:

1. Multiplique o vetor D_k pela transposta da matriz de pesos W_k , W_k' .
2. Multiplique cada componente do produto anterior pela derivada da função de ativação F para o correspondente neurônio da camada oculta.

Considere que o operador $\$$ indica a multiplicação, componente a componente, de dois vetores. O_j representa o vetor de saída da camada j e I o vetor onde todos os seus componentes são 1. Então, simbolicamente tem-se:

$$D_j = D_k W_k' \$ [O_j \$ (I - O_j)] \quad (12)$$

3.10.3 Momentum

O *Momentum* [42] é um método que visa a adição de um termo (Θ) ao ajuste de pesos, proporcional à quantidade da variação de peso anterior. Desta forma, as equações de ajuste passam a ser:

$$\Delta w_{pq,k}[n+1] = \eta(\delta_{q,k} \text{OUT}_{qj}) + \Theta(\Delta w_{pq,k}[n]) \quad (13)$$

e

$$w_{pq,k}[n+1] = w_{pq,k}[n] + \Delta w_{pq,k}[n+1] \quad (14)$$

O uso do *Momentum* melhora a estabilidade da rede, uma vez que suaviza as oscilações em torno da superfície do erro. Valores típicos do coeficiente Θ giram em torno de 0.9.

4. MODELAGEM E PROJETO ORIENTADO A OBJETOS

Modelagem e projeto orientado a objetos é uma maneira de estruturar o conhecimento a cerca de um problema. A estrutura fundamental é o objeto que é combinação de uma estrutura de dados e o seu comportamento em uma única entidade.

A OMT é uma metodologia de modelagem e projeto orientado a objetos. Ela aplica-se desde a fase de análise até o projeto de um sistema. Esta metodologia abstrai três aspectos do domínio do problema, sendo as características estática, comportamental e funcional. Cada um destes aspectos dá origem a três diagramas: classes, estados e fluxo de dados.

4.1 Módulos como Classes de Objetos

Como defende Meyer [16], **encapsulamento, modularidade, extensibilidade** são características entre outras, da construção de software orientado a objetos. Para Meyer, cada módulo é uma classe de objetos que apresentam características comuns. Cada objeto da implementação é uma abstração do objeto do mundo real, que procura modelar pelo menos, as características de interesse para a implementação.

As características dos objetos se referem as operações e atributos que estes podem ter. Um módulo ou classe possui uma interface pública, em que os clientes da classe (objetos de outras classes) podem se comunicar com os objetos desta classe. Neste sentido, uma mensagem de um objeto para outro objeto é realizada através de um método de sua interface pública[02][16][44].

4.2 Modelagem e Projeto Orientado a Objetos

Rumbaugh [24] vai mais longe e define uma metodologia de análise e desenvolvimento orientado a objetos, a *Object Modeling Technique (OMT)*. Desta forma, desde a análise até a implementação, usa-se uma semântica uniforme, onde todas as entidades em cada fase são a mesma: classes com seus atributos e operações.

A OMT procura modelar um sistema sob três aspectos, cada um produzindo três modelos:

- **Modelo Estático** - Procura-se identificar as classes dos objetos que compõem o sistema. Cada classe é composta de atributos e operações que seus objetos podem realizar. Para a descrição estática, utiliza-se o diagrama de classes. Neste diagrama são identificados as classes dos objetos, seus atributos, suas relações, operações e a hierárquica entre as classes.
- **Modelo Dinâmico** - consiste na descrição do comportamento dos objetos, apresentando a seqüência de estados e ações possíveis dos objetos como respostas a eventos que ocorrem no sistema. A máquina de estados é o diagrama usado para a descrição do comportamento dos objetos. O modelo dinâmico captura o aspecto de controle do sistema, sem considerar o que as operações fazem, como elas operam ou como elas são implementadas.
- **Modelo Funcional** - é descrição das transformações possíveis nos objetos durante a realização das suas operações. Para tanto, se utiliza do diagrama de fluxo de dados. O modelo funcional, captura os aspectos de como o sistema faz, desconsiderando para quem ou quando é feito.

4.3 Modelo Estático

O diagrama de classes fornece uma notação gráfica para a modelagem de objetos, classes e suas relações. Através dele, pode-se observar mais claramente os aspectos relacionados a natureza estática dos objetos do domínio do problema.

4.3.1 Classes

A figura 4.1 mostra a notação de modelagem OMT para classes. A classe é representada por uma caixa contendo três divisões: a primeira contém o nome da classe, a segunda enumera os atributos e a terceira divisão, enumera as operações da classe.

Na figura 4.2 é mostrada a representação das classes Pessoa e Arquivo. Na figura pode-se notar que a classe Pessoa tem os atributos nome e idade, enquanto a classe Arquivo, contém os atributos nome, tamanho e data da última atualização. Quanto às operações, a figura 4.2 indica que a classe Pessoa contém as operações *troca-de-trabalho* e *troca-de-endereço* e, a classe Arquivo, a operação *gravar*.

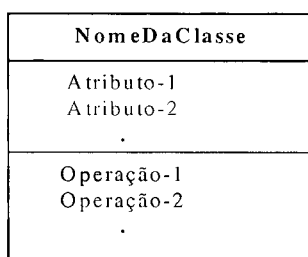


Figura 4-1 Notação de uma Classe no Diagrama de Classes

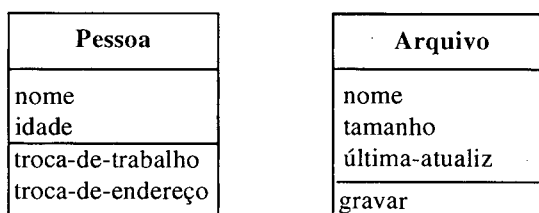


Figura 4-2 Classes Pessoa e Arquivo

4.3.2 Herança

Herança é a relação entre uma classe e uma ou mais versões especializadas desta classe [05] [16]. A classe menos especializada é chamada de superclasse ou classe ancestral. As classes

especializadas são chamadas de subclasses ou classes derivadas. Por exemplo, Aluno e Professor são subclasses da classe Pessoa.

Cada subclasse herda as características da superclasse. A subclasse, entretanto, pode conter características que sua superclasse não possui, por exemplo, um atributo ou uma operação à mais. No exemplo, a classe Aluno herda o atributo endereço da classe Pessoa.

A relação de herança entre duas classes é representada, no diagrama de classes, como um triângulo. A figura 4.3 mostra a relação de herança entre as classes Pessoa, Aluno e Professor. Nota-se que nas subclasses não são redefinidas as características constantes na superclasse, pois estas são naturalmente herdadas pelas classes derivadas.

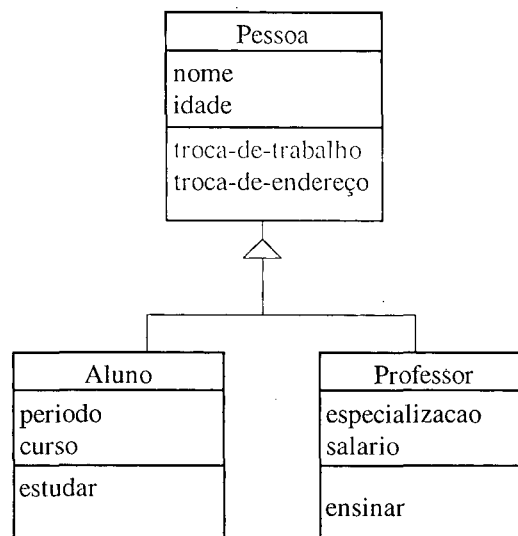


Figura 4-3 Relação de Herança Entre as Classes Pessoa, Aluno e Professor

4.3.3 Associação e Ligação

Uma ligação é uma conexão conceitual ou física entre instâncias de objetos [24]. Por exemplo, João *trabalha-para* a companhia Orange. Matematicamente, uma ligação é definida como uma tupla, ou seja, uma lista de objetos.

Um grupo de ligações, com uma estrutura e semântica comuns, forma uma associação. *Trabalha-para*, no exemplo anterior, forma uma associação, pois pode-se ter outras ligações como José *trabalha-para* o governo e Lucia *trabalha-para* o governo.

A multiplicidade de uma associação especifica quantas instâncias de uma classe podem relacionar-se com uma simples instância de uma classe associada. A multiplicidade restringe o número de objetos relacionados. Ela é freqüentemente descrita como sendo de “um” ou “muitos” objetos relacionados. Por exemplo, a associação *trabalha-para* pode ser uma associação de um-para-muitos, se um trabalhador somente pode trabalhar em uma empresa ou, uma relação muitos-para-muitos se um trabalhador pode trabalhar em várias empresas.

A figura 4.4 descreve a notação usada para representar as associações entre classes.

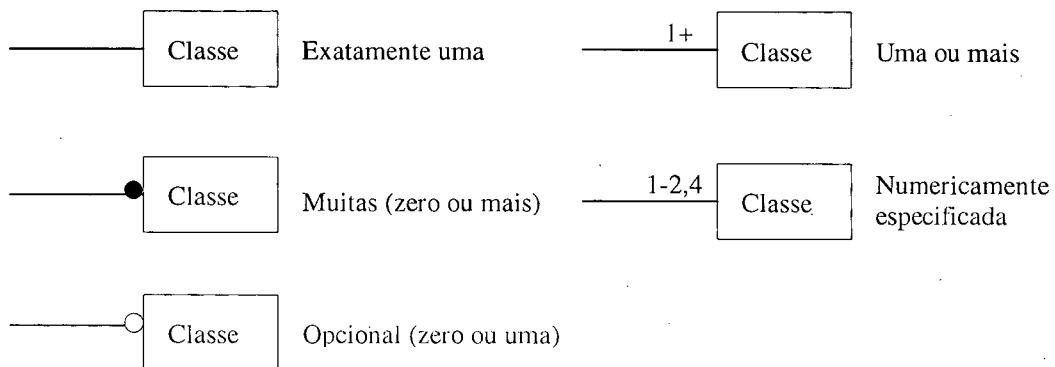


Figura 4-4 Multiplicidade de Associações

A figura 4.5 representa a associação *trabalha-para* entre a classe Pessoa e a classe Empresa, uma associação muitos-para-muitos. Pode-se notar que o nome da associação é colocado em cima da linha que a descreve.

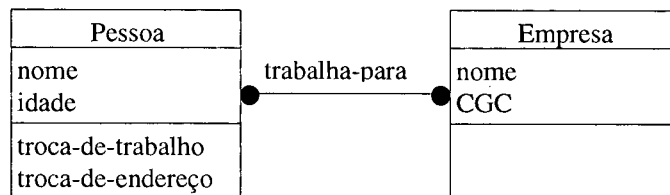


Figura 4-5 Associação trabalha-para

4.3.4 Agregação

Agregação [24] é a relação “parte-todo” ou “parte-de” nas quais objetos representam os componentes de algum outro objeto, ou seja, um objeto é agregação de outros.

A figura 4.6 mostra a notação usada para a relação de agregação e a figura 4.7 exibe um exemplo de um documento composto de vários parágrafos.

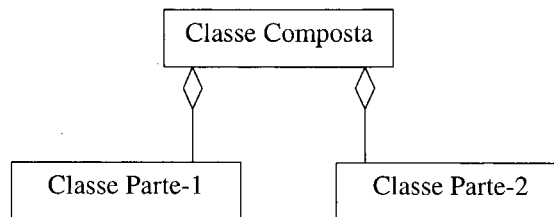


Figura 4-6 Notação da Relação de Agregação

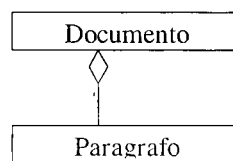


Figura 4-7 Exemplo de Agregação

4.4 Diagrama de Estados

O controle de fluxo é o aspecto de um sistema que descreve as seqüências de operações que ocorrem em resposta a uma estímulo externo. O conceito de modelagem dinâmica envolve a noção de eventos, representando estímulos externos, e estados, representando os valores dos objetos.

Um evento é algo que acontece em um determinado momento do tempo, por exemplo, o pressionar de uma tecla ou o lançamento da Apolo 11.

Um estado é uma abstração dos valores dos atributos e ligações de um objeto. Por exemplo, uma empresa pode estar em estado de falência, significando que os valores de seus débitos exigem que a empresa encerre suas atividades.

Um diagrama de estados é um grafo cujos nós são estados e os arcos direcionados, representam eventos que provocam transições entre estados. A figura 4.8 mostra a notação usada para representar estados e eventos. Pode-se notar que um estado é representado por uma caixa com cantos arredondados, contendo o nome do estado e uma afirmação a qual indica uma

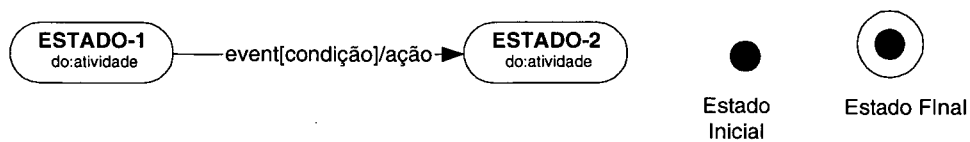


Figura 4-8 Notação de Estados e Eventos

atividade que ocorre quando o objeto se encontra neste estado. Uma atividade pode ser uma operação contínua como a exibição de uma figura ou uma seqüência de operações que tem um início e término. Uma atividade pode ser também expandida em um outro diagrama de estados para ser melhor detalhada. A figura 4.9 exibe um exemplo de uma diagrama de estados de um teclado numérico, como o de um telefone.

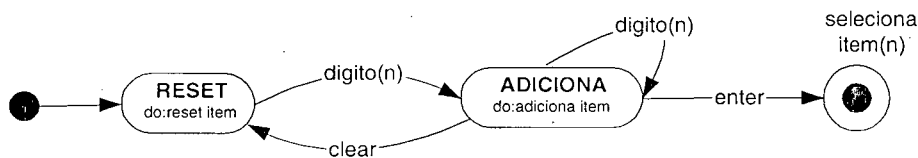


Figura 4-9 Diagrama de Estados de um Teclado Numérico

4.5 Diagrama de Fluxo de Dados

Um diagrama de fluxo de dados mostra as relações funcionais entre valores calculados por um sistema, incluindo valores de entrada, valores de saída e valores de dados intermediários [24].

Um diagrama de fluxo de dados é um grafo mostrando o fluxo de valores dos objetos das suas origens, passando através de processos que os transformam em outros objetos, até os seus destinos.

4.5.1 Processos

Um processo é algo que transforma valores. Processos de baixo nível são funções puras como a divisão de dois números ou as trocas financeiras envolvidas nas transações de um cartão de crédito. A representação de um processo é mostrada na figura 4.10. Um processo é representado por uma elipse contendo o nome do processo. O processo pode ter valores que

entram, representados por arestas com uma seta apontado para o seu interior e, valores que saem do processo, representados por arestas que apontam para fora.

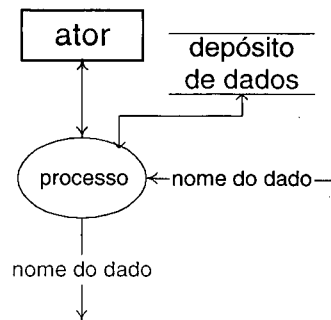


Figura 4-10 Representação de Processos, Atores e Depósito de dados

Uma caixa pode representar um objeto fornecendo ou recebendo algum valor do processo, ou ainda, alguma transformação efetuada pelo processo sobre o objeto.

Um exemplo é dado na figura 4.11 do processo que envolve a divisão de um número.

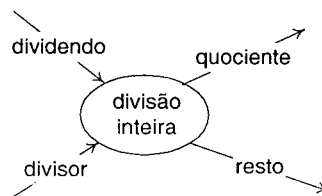


Figura 4-11 Processo da Divisão de um Número Inteiro

4.5.2 Atores

Um ator é um objeto ativo que direciona o fluxo de dados produzindo ou consumindo valores. Exemplos de atores incluem um termostato, um motor sob controle de um computador ou uma interface de rede.

Um ator é desenhado como um retângulo, indicando que ele é um objeto, como mostra a figura 4.10. A figura 4.12, mostra o exemplo simplificado do diagrama de fluxo de dados da tabela periódica.

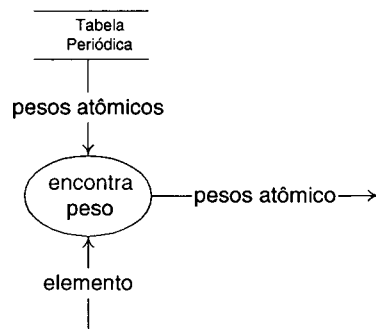


Figura 4-12 Diagrama de Fluxo de Dados da Tabela Periódica

4.5.3 Depósito de Dados

Um depósito de dados é um objeto passivo no diagrama de fluxo de dados, armazenando dados para serem usados mais tarde. Os depósitos de dados são objetos diferentes dos atores, pois não originam operações, mas puramente respondem a pedidos para armazenagem e acesso de dados.

Exemplos de depósito de dados são listas, tabelas e base de dados. Na figura 4.10, a representação de um depósito de dados é mostrado como duas retas paralelas com o nome do depósito entre elas. No exemplo da figura 4.12, a tabela periódica é mostrada como um depósito de dados, armazenando elementos químicos e seus respectivos números atômicos.

5. AGENTE ADAPTATIVO

Uma das características mais marcantes do ser humano é a da **adaptação** às mudanças que ocorrem no ambiente. Esta característica é devida principalmente ao fato que a capacidade de aprendizagem é inerente ao ser humano.

Neste sentido, o homem leva extrema vantagem sobre a máquina, mesmo em relação àquelas que são construídas para realizarem tarefas complexas como os autômatos, pois não são adaptativos. As mudanças de estados que ocorrem no contexto a que estão inseridos os autômatos, devem ser previstos pelo seu programa.

Algo semelhante acontece com as aplicações de gerência de redes de computadores. A maioria são autômatos que simplesmente respondem com uma ação, prevista algoritmicamente, a eventos ocorridos nos objetos gerenciáveis. Para eventos extraordinários e não previstos, o gerente da rede, o ser humano, deve tomar a decisão adequada.

As redes neurais, de um modo mais restrito, também podem apreender e portanto, podem se adaptar às mudanças de contexto. Desta forma, a construção de aplicações de gerência usando redes neurais, pode originar gerentes e agentes que podem se adaptar diante das constantes mudanças que ocorrem na forma de eventos numa rede de computadores.

5.1 Modelos de Gerência Não-Adaptativos

Um modelo de gerência, objeto de estudo do capítulo 2, geralmente se compõe de agentes e gerentes.

O agente controla um conjunto de objetos de um nó gerenciável, enquanto que o gerente é responsável por realizar as funções de gerência, monitorando e controlando os objetos

gerenciáveis, através de pedidos enviados aos agentes e *traps* enviados pelos agentes para o gerente, conforme descreve a figura 5.1.

Neste modelo, tanto os agentes como o gerente, geralmente não são adaptativos.

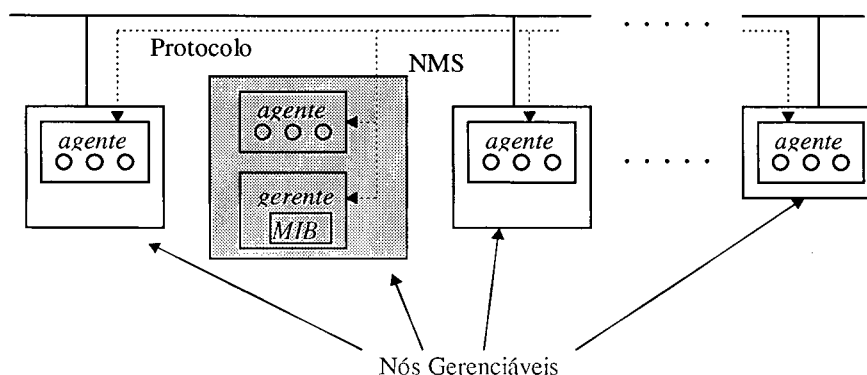


Figura 5-1 Modelo de Gerência

5.2 Implementação da Adaptação ao Modelo de Gerência

Numa primeira análise, surgem três possibilidades de implementação de adaptação no modelo de gerência tradicional, resumidos pelo quadro 5.1:

- **Implementar um modelo onde somente os gerentes são adaptativos.**

Esta implementação é a mais natural, pois a política de gerência é implementada pelos gerentes. Qualquer decisão, provocada por um evento extraordinário, deve estar de acordo com a política adotada pela administração da rede.

Esta implementação está centralizada no gerente e todos os agentes existentes podem ser reutilizados. Além disso, o tráfego na rede pode influenciar a comunicação entre o agente e o gerente, tornando-a lenta.

- **Implementar um modelo onde somente os agentes são adaptativos**

Esta implementação tem a vantagem de ser mais eficiente. As decisões são tomadas localmente, sem precisar enfrentar o tráfego da rede e esperar pelo atendimento do gerente. Em algumas situações críticas, pode ser uma vantagem extremamente desejável.

Nesta implementação, parte da política de gerenciamento fica distribuída pelos agentes. Assim, as mudanças são mais difíceis, pois pode haver a necessidade de mudanças em todos os agentes.

- **Implementar um modelo onde tanto o gerente como os agentes são adaptativos.**

Tomando-se cuidado, com a implementação de agentes e gerentes adaptativos pode-se obter uma solução com ganho de eficiência em situações críticas, mas com partes da política de gerência, estratégicas e sujeitas a constantes mudanças, implementadas pelo gerente. O agente se adapta em questões sobre sua autonomia.

Implementação	Tráfego na rede	Reutilização de agentes	Localização da Política de gerência
somente com gerente adaptativo	alto	sim	influencia na centralização
somente com agentes adaptativos	baixo	não	influencia na distribuição
gerente e agentes adaptativos	médio	não	flexível

Quadro 5-1 Características da Implementação da Adaptação ao Modelo Tradicional

Além disso, esta solução não permite que sejam reutilizados os agentes existentes, pois estes teriam que ser reescritos para incluir o código implementando a parte adaptativa.

5.3 Extensão Adaptativa de um Agente

Como foi visto, há três formas para implementar a parte adaptativa em um modelo de gerência de redes, cada uma com suas vantagens e desvantagens descritas no quadro 5.1. Todas

estas formas procuram implementar a característica adaptativa, alterando-se o código do gerente ou agente.

Entretanto, o que se deseja é estender o gerente ou o agente sem a necessidade de reescrevê-los, o que muitas vezes é até impossível. Portanto, a reutilização do agente e do gerente é um requisito muito desejado.

No Padrão de Gerência Internet, a forma de estender os agentes é dada pelo protocolo SMUX. O enfoque passa a ser a criação de um subagente que irá implementar a característica adaptativa. Na ocorrência de um evento, em que o subagente consiga tomar uma decisão, ele não é repassado ao gerente, em vista que uma ação pode ser tomada pelo próprio subagente, conforme descreve a figura 5.2.

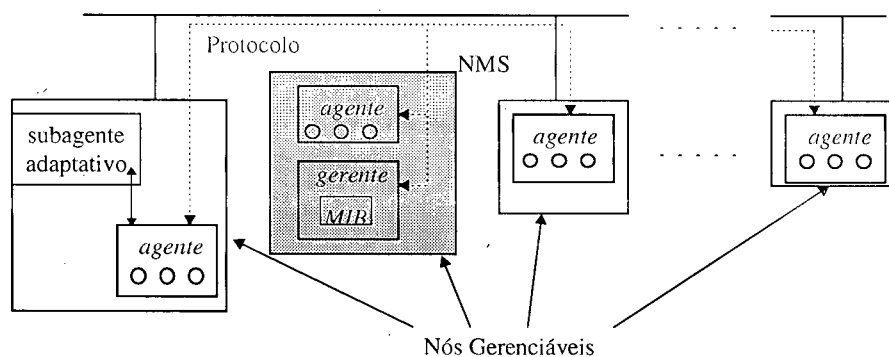


Figura 5-2 Um Subagente Adaptativo

Este módulo implementa a parte adaptativa necessária do agente. Importante notar que o agente do fornecedor é conservado intacto, sendo que ele é estendido com a característica adaptativa fornecida pelo subagente, tornando-se um **agente adaptativo**.

5.3.1 Características do Agente Adaptativo

Um agente adaptativo introduz as seguintes características:

- **Diminuição do tráfego entre o gerente e o nó gerenciável**

O tráfego entre o gerente e o nó gerenciado é otimizado, sendo que o agente adaptativo age como um filtro entre os dois.

Um exemplo é a ocorrência de um *trap* onde o agente adaptativo pode tomar uma decisão a cerca do evento. Neste caso, não há a necessidade de repassar o *trap* para o gerente, o que evita o envio de mensagens SNMP pela rede.

- **O gerente passa a ver os objetos gerenciáveis sob uma maior abstração**

Devido ao fato que muitas decisões podem ser tomadas pelo agente adaptativo e portanto, alguns atributos e, até mesmo o conhecimento completo de alguns objetos gerenciáveis passam a ser desnecessários.

O nó gerenciável desta forma, pode ser visto pelo gerente sob uma visão mais simples ou com um nível de abstração maior.

- **Tomada de decisões mais rápidas**

Se algum evento ocorrer, como um alerta, requerendo uma ação rápida e que o agente adaptativo possa tomar, então haverá mais chances do problema ser solucionado. De outra forma, em uma rede congestionada, por exemplo, o gerente poderia tomar a decisão tarde demais.

- **Adaptação**

Um agente adaptativo está preparado para todas as mudanças de ambiente em que foi treinado. Por exemplo, pode-se treinar um agente adaptativo para tomar certas decisões que o administrador de rede tomava na console da aplicação gerente.

Com um agente adaptativo, o nó gerenciável passa a adquirir alguma autonomia com relação ao gerente, principalmente em questões não críticas. Desta forma, a gerência da rede torna-se mais automatizada.

5.3.2 Funcionalidade

A figura 5.3 descreve a extensão ao agente SNMP local, formando o Agente Adaptativo. Esta extensão é a implementação de um subagente SMUX, composto pelos módulos Rede Neural, Interface SMUX e vários Objetos Gerenciáveis.

O Rede Neural monitora os Objetos Gerenciáveis, lendo vários atributos. Os valores destes atributos servem como entradas que serão processadas pela Rede Neural, resultando em várias saídas enviadas aos Objetos Gerenciáveis, indicando ações ou novos valores de atributos que devem ser tomados.

A Interface SMUX serve como elo de comunicação do subagente com o agente SNMP. Ela realiza o estabelecimento de associação, efetua o registro de subárvores privadas e recebe e envia mensagens SNMP.

5.3.3 Visão Fornecida por um Agente Adaptativo

Um agente adaptativo pode fornecer uma visão mais abstrata dos objetos da MIB implementada pelo seu agente SNMP, obrigando a que sejam feitas novas definições de objetos na MIB do gerente. Isto não chega a preocupar, diante do fato de que escrever uma definição de objeto gerenciado em ASN.1 é muito mais simples do que escrever um novo agente.

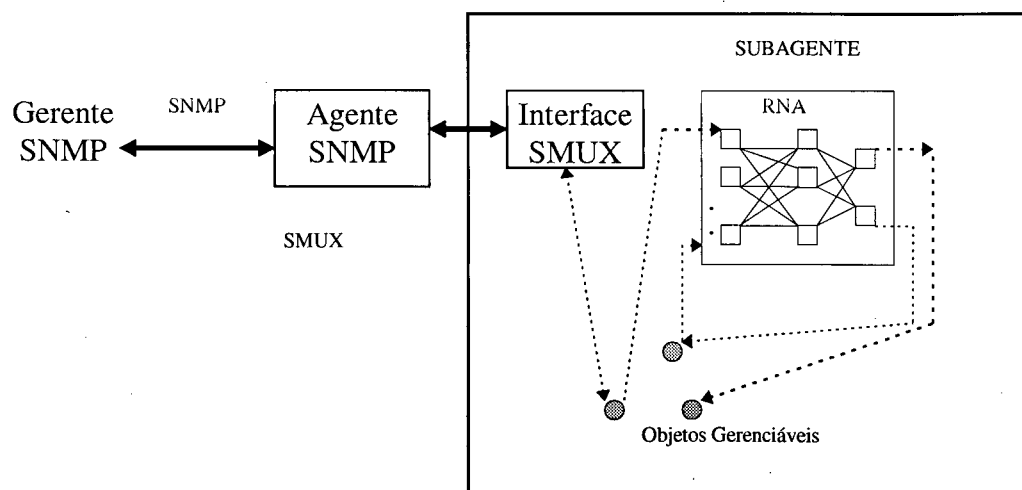


Figura 5-3 Implementação do Agente Adaptativo com Redes Neurais

5.3.4 Treinamento de um Agente Adaptativo

Um agente adaptativo, antes de estar pronto para operar como *daemon*, deve ser treinado. O treinamento deve ser realizado com um conjunto de treinamento composto por pares de vetores. Cada par de vetores de treinamento deve conter um vetor de entrada e um vetor de saída desejada. O vetor de entrada representa os possíveis eventos que podem ocorrer e o vetor de saída desejada, as possíveis ações a serem tomadas para cada evento.

Este conjunto de treinamento é armazenado em uma base de dados, sendo que somente é consultada pelo agente adaptativo na fase de treinamento. Portanto, toda vez que for alterada esta base, novo treinamento deve ser efetuado.

6. AMBIENTE PARA O DESENVOLVIMENTO DE AGENTES ADAPTATIVOS (ADAA)

As metodologias de desenvolvimento e projeto orientadas a objetos estão em rápida expansão e já existem diversas ferramentas no mercado que as implementam. Basicamente todas tratam muito bem a fase de análise e algumas indo mais além, tratam a fase de projeto, gerando por exemplo, código em uma linguagem de programação como C++ [38].

Para o desenvolvimento de agentes adaptativos, a fase de análise pode ser muito bem resolvida por algumas destas ferramentas. Contudo, é na fase de projeto que se encontram as maiores dificuldades. Não existem ferramentas adequadas que tornem eficiente a construção até mesmo de agentes não-adaptativos para o modelo Internet. Mesmo a disponibilização de bibliotecas de classes que encapsulem os conceitos de gerência de redes não são encontradas.

Portanto, o desenvolvimento orientado a objetos de agentes adaptativos, particularmente utilizando a OMT, deve ser apoiado com a construção de um suporte mínimo para que a fase de projeto não seja tão onerosa e difícil.

6.1 Domínio do Problema

A implementação de um agente adaptativo para gerenciar objetos gerenciáveis pode ser resumida a adição da característica adaptativa a um agente existente, tornando possível que este possa tomar determinadas decisões. Além disso, o agente deve continuar sendo controlado também pelo gerente da rede através do protocolo de gerência. Portanto, dois problemas devem ser analisados:

- **Extensão do Agente**

Muitos fornecedores de Plataformas de Gerência SNMP baseam seus produtos no OpenView da HP. A DEC, IBM e AT&T/NCR utilizam parte do código da HP em seus produtos, tendo em consequência, muitos elementos em comum.

Um desses elementos, é o fornecimento de uma biblioteca de funções projetada por Marshall Rose [11], a qual implementa a interface SMUX. Esta biblioteca possibilita a construção de subagentes e apesar de ser um avanço, apresenta alguns problemas:

- **Baixa reutilização de software:** todo o subagente deve ser praticamente escrito, sem poder aproveitar alguma implementação já pronta. No máximo, o que se faz, é desenvolver cada subagente em cima de um código padrão que serve como esqueleto ou guia para os programadores.
- **Baixa abstração:** o usuário da biblioteca deve conhecer muito bem os conceitos envolvidos no padrão Internet. Muito do código do subagente é devido às características e regras dos protocolos SNMP e SMUX. Além disso, o programador deve ter uma boa noção da MIB Internet, principalmente no que se refere a nomenclatura das variáveis.

Portanto, esta solução apesar de permitir a construção de subagentes, ainda exige muito esforço do implementador e não parece ser a mais adequada. Entretanto, esta solução é a única conhecida e que traz várias implementações em diferentes plataformas.

- **Implementação da Característica Adaptativa**

Em gerência de redes, a utilização de redes neurais ainda é muito recente e ainda, à nível de pesquisa. Existem muitas ferramentas para testes e construção de redes neurais, mas todas são de difícil integração com um ambiente de gerência de redes, ainda mais para a construção de agentes adaptativos.

6.2 Objetivos

O principal objetivo do **Ambiente de Desenvolvimento de Agentes Adaptativos (ADAA)** é facilitar o desenvolvimento de agentes adaptativos utilizando redes neurais, ou seja, tornar rápida, eficiente e segura a construção de agentes adaptativos.

Como definido no contexto do problema, os implementadores que utilizarão o sistema serão, basicamente, programadores acostumados a utilizar funções de gerência fornecidas por bibliotecas.

Desta forma, os seguintes objetivos devem ser pretendidos:

- **Encapsulamento das Teorias Suportes** - O desenvolvimento deve permitir a construção de agentes adaptativos sem que o implementador tenha muito conhecimento da teoria de redes neurais ou gerência de redes de computadores. O implementador deve especializar-se, no máximo, em conhecer o contexto do problema no qual o agente adaptativo irá atuar.
- **Modularidade** - A implementação deve apresentar-se como um conjunto de módulos de sistemas, tornando fácil a compreensão geral do contexto do problema. Cada módulo deve descrever um componente do sistema que modela um conjunto de objetos do domínio do problema. A intenção aqui é reduzir, ao máximo, a distância semântica entre os objetos do problema e os objetos usados na implementação. Referindo-se à implementação do agente adaptativo, a modularidade deve tornar possível a compreensão geral da teoria de redes neurais e gerência de redes envolvidas na solução de um determinado problema. Esta compreensão deve ser possível através da interface de cada módulo, a qual torna público determinado conhecimento a cerca do objeto modelado.
- **Extensibilidade** - A implementação deve seguir o princípio **aberto-fechado** [09]:
 1. Os módulos devem poder ser estendidos. Desta forma, a um módulo pode ser adicionado novas características para modelar determinados objetos. Geralmente a implementação é baseada em objetos genéricos, mas um determinado uso pode necessitar de objetos que, além das características genéricas modeladas pela implementação, apresenta características específicas que dizem respeito ao contexto do problema.

2. Cada módulo deve estar imediatamente pronto para ser usado.

- **Prototipação da Rede Neural** - há a necessidade de uma ferramenta de prototipação de redes neurais, onde se pode testar vários tipos e configurações de redes neurais, procurando encontrar as mais adequadas ou até mesmo, servindo simplesmente para validar a solução utilizando redes neurais.

6.3 Descrição do Ambiente

A implementação se destina ao desenvolvimento orientado a objetos de agentes adaptativos. Neste sentido, o implementador de um agente adaptativo, deve usar uma linguagem de programação que suporte este paradigma, preferencialmente o C++.

Essencialmente, o que o ambiente fornece são bibliotecas que implementam classes que modelam conceitos de gerência de redes Internet e de redes neurais, e também, um programa de prototipação e treinamento de redes neurais.

A **Biblioteca de Classes de Gerência de Redes Internet - SNMPLib**, por ser a maior, é a mais extensa e complexa. Esta biblioteca, destina-se a fornecer classes para a construção de subagentes, adaptativos ou não, sendo uma alternativa orientada a objetos da biblioteca apresentada por Marshall Rose.

A **Biblioteca de Classes de Redes Neurais - Neurolib**, tem por objetivo fornecer vários tipos de redes neurais (no momento, somente fornece a rede *Backpropagation*). Esta biblioteca, da mesma forma, pode ser usada para implementar outros tipos de programas que não sejam agentes adaptativos.

O **Programa para Prototipação e Treinamento de Redes Neurais - NeuroProto**, foi desenvolvido para facilitar a prototipação do agente adaptativo. Com este programa, o implementador pode treinar e testar a convergência de sua rede com seu conjunto de treinamento. Uma vez treinada e testada a rede, o implementador pode salvar as sinapses da rede neural para ser usada no agente adaptativo. Isto também permite que o implementador refine periodicamente o treinamento do agente adaptativo, trocando seu treinamento como achar conveniente.

6.4 SNMPlib

6.4.1 Modelo da Biblioteca da SNMPlib

O modelo em que se baseia a SNMPlib é o modelo Agente-Gerente Internet exibido na figura 6.1.

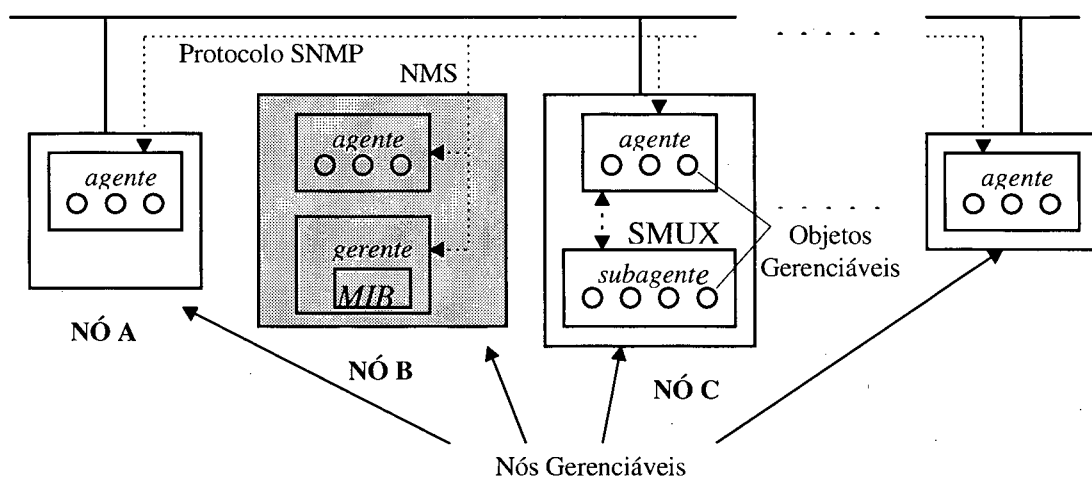


Figura 6-1 Modelo Agente-Gerente com Subagente

Como mostra a figura, os agentes se comunicam com o gerente através do protocolo SNMP. Cada agente controla um conjunto de objetos gerenciáveis, sendo que o gerente envia pedidos para o agente atuar nos objetos gerenciados. Da mesma forma, o agente pode enviar notificações sobre os objetos gerenciáveis sob seu controle ao gerente.

No nó C, é mostrado um caso especial, onde um subagente se comunica com o agente local, através do protocolo SMUX. Um subagente é uma maneira de estender a funcionalidade de um agente SNMP, tornando possível a implementação de novos processos para controlar outros objetos gerenciáveis, além daqueles controlados pelo agente SNMP.

O implementador necessita, basicamente, escrever o código para tratar os pedidos SNMP recebidos do gerente e para enviar a este, possíveis notificações ou alarmes (*traps*).

6.4.2 Conceituação Básica

O desenvolvimento de subagentes é muito simplificado com o uso das classes da SNMPlib. Na grande maioria das vezes, o usuário necessita usar somente quatro ou cinco classes básicas para modelar:

1. Objetos gerenciáveis
2. Tipos de objetos
3. Subagentes
4. Sintaxes de objetos
5. *Traps*

Com o uso de instâncias destas ou de classes derivadas, pode-se construir subagentes com toda a funcionalidade que o modelo Internet permite. Todo o encargo das funções de suporte à comunicação subagente-agente são encapsuladas nas classes da SNMPlib. Algumas destas funções são:

- estabelecimento da conexão com o agente,
- recepção de mensagens,
- envio de *traps*,
- encerramento de conexão,
- registro de subárvores,
- verificação de corretude de PDU,
- verificação do tipo de objeto.

• **Objetos Gerenciáveis Definidos pela SMI e pela SNMPlib**

Todas as características gerenciáveis de um objeto computacional que modela um recurso de rede, são vistas no modelo Internet como objetos gerenciáveis [19]. Exemplos destas características podem ser um atributo estático como a versão e *release* de um sistema operacional ou, um atributo que quando modificado, provoca a alteração ou uma ação em um recurso.

Na SNMPlib, o conceito de objeto gerenciável da SMI da Internet é realizado como sendo um atributo da classe *sMO*, a qual implementa um objeto gerenciável. Este atributo é uma

instância da classe *sOT* e implementa todas as características da macro *ASN-1 Object-Type* para definir tipos de objetos, tais como sua sintaxe e identificador de objeto.

A classe *sMO* define os métodos abstratos *procGet* e *procSet* que devem ser fornecidos pelas classes derivadas. Estes métodos implementam os procedimentos para realizar as operações *Get* e *Set* respectivamente, nos atributos gerenciáveis do objeto.

- **Subagente**

O subagente, no âmbito da *SNMPLib*, controla instâncias de objetos gerenciáveis implementados pelo usuário. O subagente estabelece a conexão com o agente do nó gerenciável sob o qual é implementado, registra os atributos dos objetos gerenciáveis e permanece executando como um *daemon* do sistema operacional, recebendo e dispatchando as mensagens enviadas pelo agente aos objetos correspondentes. Tanto o agente como o subagente podem, à qualquer momento, fechar a conexão.

A figura 6.2 mostra que uma mensagem SNMP *Get* pode ser enviada do agente ao subagente. O subagente recebe e desencapsula a mensagem, separa os identificadores de objeto e chama o método *procGet*, para cada uma das instâncias dos objetos gerenciáveis *mo1*, *mo2*, etc.

Cada método *procGet*, chamado para cada instância de objeto gerenciável, recebe como parâmetros o identificador e o tipo de objeto correspondente. No método, é retirada a parte referente à raiz da subárvore privada do identificador. Na figura 6.2, esta operação é realizada pelo método *diff* sobre o objeto *o* para obter o seu identificador de objeto.

Em seguida é verificado o primeiro subidentificador de *o*. Se este subidentificador representar um tipo de objeto simples, então é chamado um método do objeto gerenciável que retorna o seu valor. Do contrário, o subidentificador pode representar um tipo construído e neste caso, é delegado ao método *procGet* do objeto gerenciável resolver esta consulta. Este processo continua recursivamente até que o identificador passado em *procGet* refira-se a um atributo simples. Quando isto ocorrer, o identificador de objeto irá ter o comprimento 1.

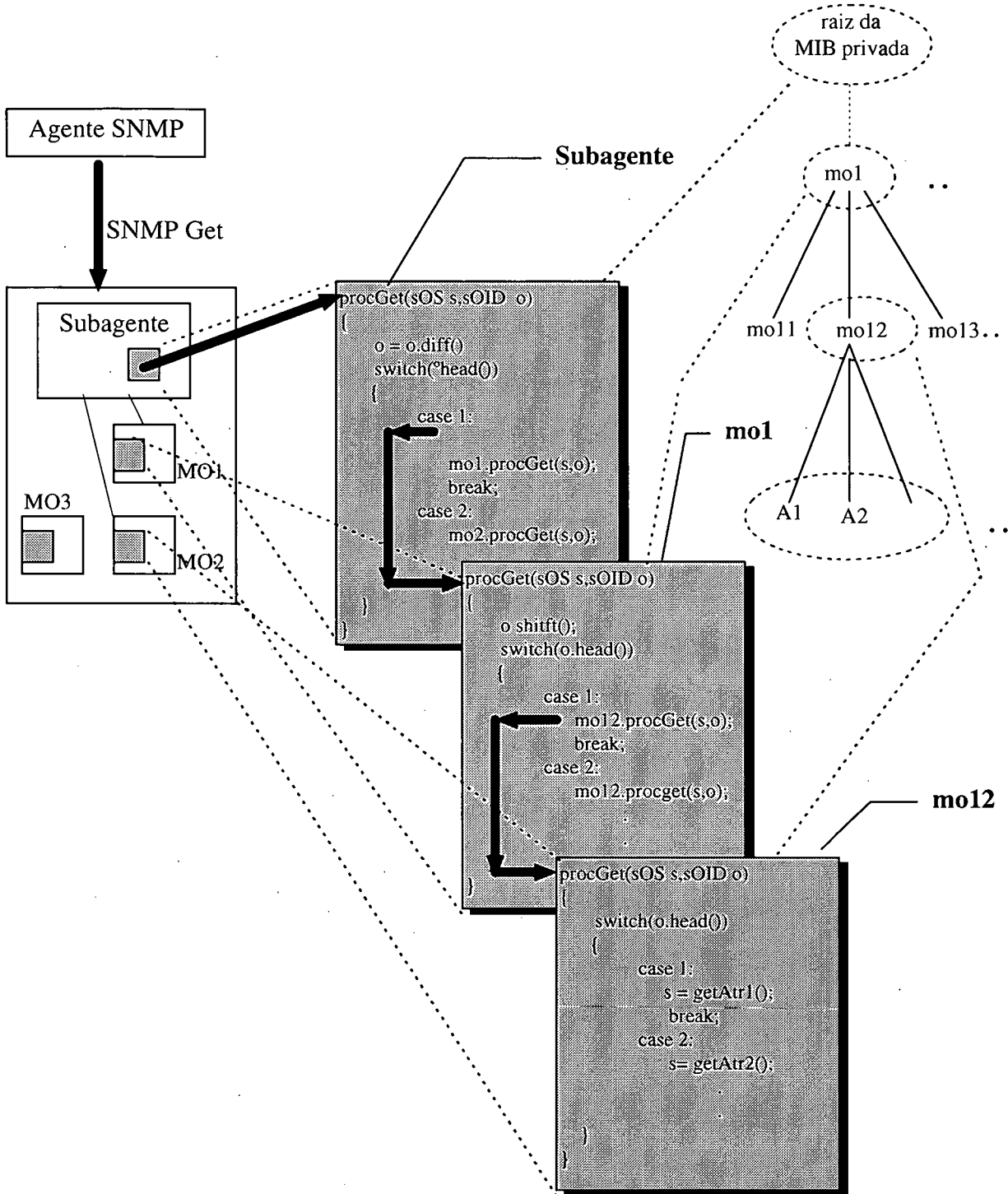


Figura 6-2 Fluxo da Operação SNMP Get em uma Instância de um Subagente

6.4.3 Definição SMI

Com as classes identificadas, incluindo seus relacionamentos de agregação, associações entre classes e identificados os atributos gerenciáveis, um subproduto que necessariamente deve surgir da fase de projeto é a definição ASN.1 dos tipos de objetos.

Algumas regras podem ajudar na construção desta definição:

- Pela forma como é construída a definição em ASN.1, a representação de relacionamentos de **agregação** é direta. Por exemplo, a figura 6.3 representa a agregação da classe *routeTable* pela classe *IP*.
- A implementação de **associações** entre classes como sugere Rumbaugh [24] pode ser conseguida utilizando-se ponteiros de classes. Na SMI, o mesmo princípio pode ser seguido, utilizando-se identificadores de objeto.
- Cada classe pode ser representada por uma macro *Object-Type*, cuja sintaxe não seja a de um tipo simples, de aplicação ou a de um grupo (*ip*). Na figura 6.4, a classe *IP* é representada por um grupo e a classe *routeTable* é representada por um tipo de objeto (*ipRoutingTable*).
- Cada atributo primitivo (por exemplo, inteiro, caracter ou *string*) de um objeto gerenciável, pode ser representado por um tipo simples ou por um tipo de aplicação. Na figura 6.4, o atributo gerenciável *destination* é definido, em ASN.1, como o atributo *ipRouteDest* do tipo *IpAddress*. Um atributo gerenciável não primitivo é um caso de agregação. O quadro 6.1 resume a correspondência entre os atributos primitivos e os tipos simples e de aplicação.

Por exemplo, a figura 6.3 descreve uma classe denominada *route* que modela um objeto gerenciável representando uma rota da camada IP [06][20]. A camada IP, modelada pela classe *IP*, pode conter uma ou mais rotas. Desta forma, entre a classe *route* e a classe *IP*, existe a relação de agregação *routeTable*.

Tipo Primitivo	Tipo Definido pela SMI
integer	INTEGER
string	OCTET STRING
OID	OBJECT IDENTIFIER
IpAddress	IpAddress
NetWorkAddress	NetWorkAddress
integer	Counter
integer	Gauge
integer	TimeTicks
string	DisplayString
N/A	NULL
N/A	Opaque

Quadro 6-1 Correspondência entre Atributos Primitivos e Tipos Simples e de Aplicação

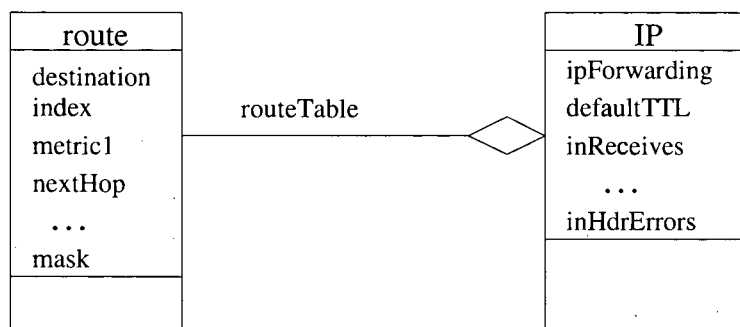


Figura 6-3 Classes *route* e *IP*

Analisando as definições ASN.1 da MIB-II, a classe *IP* pode ser associada ao grupo *IP*, a classe *route* está relacionada à sintaxe *IpRouteEntry* e por fim, a agregação *routeTable* associa-se ao tipo *ipRoutingTable*. A figura 6.4, retirada do RFC1158, descreve resumidamente estas classes.

Com a construção da definição ASN.1 que descreve o módulo da MIB privada do subagente, pode-se gerar um arquivo. Esta definição pode ser transportada e carregada no gerente, que toma assim, conhecimento da existência desta MIB.

```

RFC1158-MIB DEFINITIONS ::= BEGIN
IMPORTS
    mgmt, OBJECT-TYPE, NetworkAddress, IpAddress, Counter, Gauge,
    TimeTicks
...

at OBJECT-IDENTIFIER ::= { mib-2 3 }
ip OBJECT-IDENTIFIER ::= { mib-2 4 }
...

ipRoutingTable OBJECT-TYPE
    SYNTAX SEQUENCE OF IpRouteEntry
    ACCESS read-write
    STATUS mandatory
    ::= { ip 2 }
...

ipForwarding OBJECT-TYPE
    SYNTAX INTEGER {
        gateway(1), -- entity forwards datagrams
        host(2), -- entity does NOT forward datagrams
    }
    ACCESS read-write
    STATUS mandatory
    ::= { ip 1 }
...

IpRouteEntry ::=
SEQUENCE {
    ipRouteDest IpAddress,
    ipRouteIfIndex INTEGER,
    ...
    ipRouteMask IpAddress
}

```

Figura 6-4 Definição na MIB-II das Classes *IP* e *route* e da Agregação *routeTable*

6.4.4 Identificação de Instâncias de Objetos Gerenciáveis

A SMI especifica que a identificação das instâncias dos objetos gerenciáveis é deixada a cargo do protocolo de gerência SNMP. Apesar que em OMT, identificadores de objetos não são necessários, pois cada objeto tem sua própria identidade, na fase de projeto devem ser considerados devido ao protocolo SNMP.

De fato, no caso do protocolo SNMP, o esquema de identificação dos objetos gerenciáveis entre as entidades de gerência (agente, gerente e subagente) é relativamente

simples: um identificador de objeto, formado com a concatenação do nome do tipo do objeto mais um sufixo. Portanto,

iso.org.dod.internet.mgmt.mib.system.sysDescr.0

identifica a primeira instância do tipo de objeto

iso.org.dod.internet.mgmt.mib.system.sysDescr

tendo em vista o número 0 acrescentado ao nome da classe.

6.4.5 Exemplo de Implementação

A construção de um subagente para controle de um processo no sistema operacional UNIX [03][11][37] será descrita como exemplo, visando esclarecer melhor as fases de implementação citadas.

- **Processo no Sistema Operacional UNIX**

Em uma máquina sob o controle do UNIX, toda vez que um programa é colocado para executar, um processo é criado. Um processo é um recurso que contém atributos como o número que o identifica, a quantidade de memória RAM ocupada e o seu estado operacional.

Os processos normalmente tem um período de execução finito, mas existem alguns que ficam em execução indefinidamente, enquanto a máquina e o sistema operacional estiverem ativos. Estes processos são conhecidos como *daemons*. Um *daemon* geralmente implementa algum serviço disponibilizado aos usuários, como um servidor de arquivos, um servidor de correio eletrônico ou servidor de impressora.

Desta forma, a implementação de um subagente que seja executado como um *daemon*, monitorando algum processo ou um conjunto deles, é de extrema valia para garantir a disponibilidade dos serviços na máquina.

- **Análise Inicial**

A análise do problema resulta, em um primeiro momento, nas seguintes classes mostradas no diagrama de classes da figura 6-5:

- *Process* : cada instância desta classe representa um processo do UNIX. Suas características são as seguintes:

Atributos:

PID: número de identificação.

state: estado em um certo momento.

cpu: percentagem de utilização da cpu da máquina.

mem: percentagem de utilização da memória.

Operações:

up : coloca em execução.

down: pára a execução.

refresh: pára e reinicializa a execução.

- *procMO*: esta classe modela um processo que pode ser gerenciado via SNMP. De fato, a sua classe herda as características da classe *sMO* e da classe *Process*. Além das características herdadas, a classe possui as seguintes características:

Operações:

procGet : implementa a operação *procGet* da classe *sMO*.

procSet : implementa a operação *procSet* da classe *sMO*.

getListOT : implementa a operação *getListMO* da classe *sMO*.

- *procmond*: esta classe implementa as características de um subagente para controlar o objeto gerenciável, instância de *procMO*:

Operações:

procGet : implementa a operação *procGet* da classe *sSagent*.

procSet : implementa a operação *procSet* da classe *sSagent*.

No diagrama de classes da figura 6.5 , as classes *sMO* e *sAgent* são representadas tracejadas, pois são classes da SNMPlib.

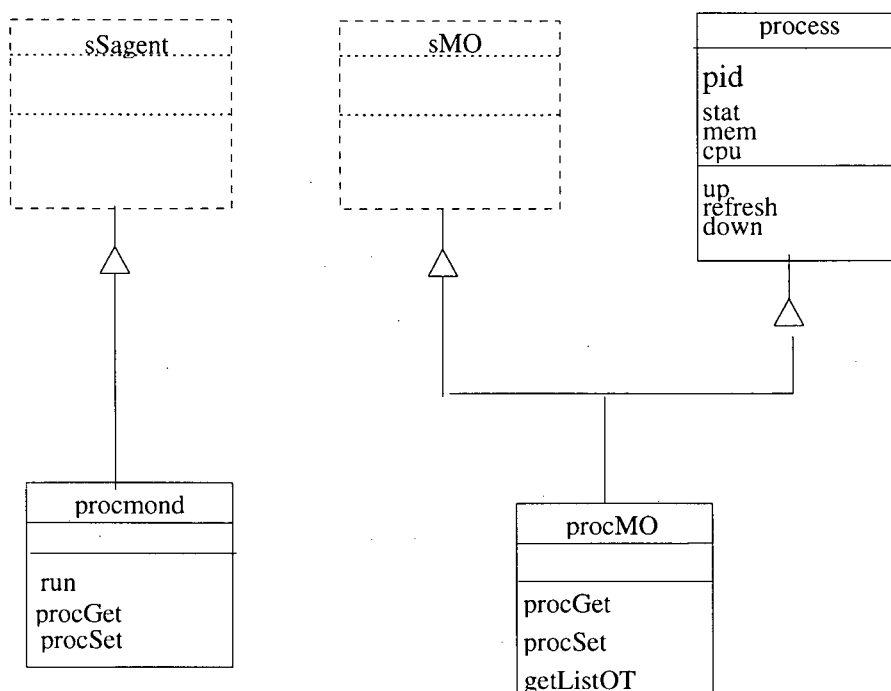


Figura 6-5 Diagrama de Classes

- Classe *procmod*

A classe *procmond* é derivada da classe *sSagent*. Pode-se notar que esta classe fornece os métodos *run*, *procGet* e *procSet*. A figura 6-6 descreve os possíveis estados de uma instância *procmod* quando cada um desses métodos é chamado. A figura 6-7 descreve o diagrama de fluxo de dados dos processo *run*, *procGet* e *procSet*. O comportamento de uma instância *procmod* pode ser descrito pela seguinte seqüência de procedimentos:

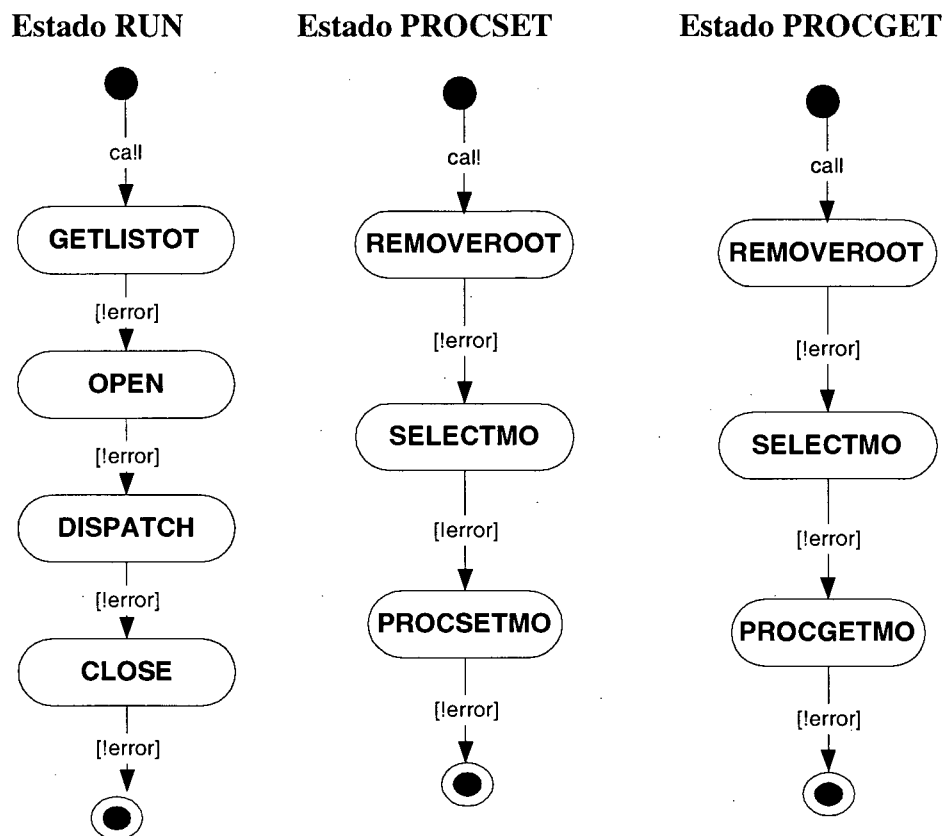


Figura 6-6 Diagrama de Estados para os Processos *ProcGet* e *ProcSet*

1. No Estado RUN, é obtido de todas as instâncias da classe SMO, as instâncias da classe sOT para formar a MIB Privada (no subestado GETLISTO).
2. Em seguida esta lista de tipos de objetos é fornecida como parâmetro para o processo *open*, que abre o módulo MIB (não representado na figura), sendo que *procm* atinge o subestado OPEN.
3. O processo *dispatch* é chamado e o *procm* atinge o subestado DISPATCH. Neste subestado, o processo *dispatch* (internamente pode chamar os métodos *procSet* e *procGet*) pode eventualmente retornar.
4. Se o processo *dispatch* retornar, o processo *close* é chamado. Desta forma, *procm* passa para o subestado CLOSE, terminando o processo *run*.

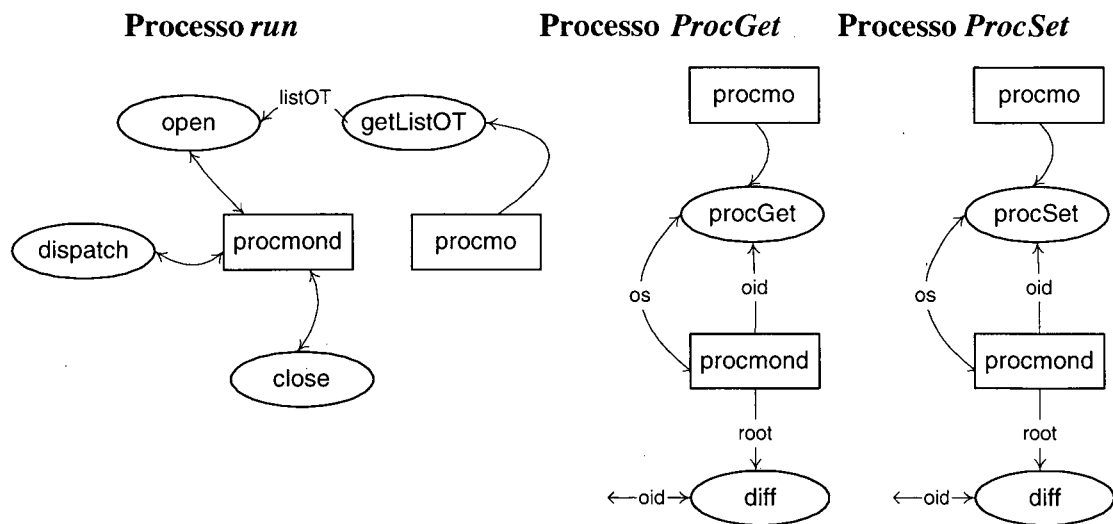


Figura 6-7 Diagrama de Fluxo de Dados para os Processos *Run*, *ProcGet* e *ProcSet*

- **Classe *procMO***

A classe *procMO* tem o diagrama de estados da figura 6.8 e o diagrama de fluxo de dados da figura 6.9. Os diagramas descrevem o seguinte comportamento de uma instância da classe *procMO*:

1. Inicialmente, uma instância tem o estado inicial DOWN. Isto significa que nenhum processo está executando na máquina.
2. Com a chamada do método *up*, um processo é criado e o estado da instância passa a ser UP, significando que o processo se encontra executando normalmente.
3. Estando a instância no estado UP, o método *refresh* pode ser chamado, fazendo com que o processo seja reinicializado, ou seja, é forçado a ir para o estado REFRESH. Em seguida, depois da operação *refresh*, é novamente posto para executar, indo para o estado UP.
4. O estado DOWN pode ser novamente atingido quando o processo não está operacional, e neste caso, nenhum serviço está sendo disponibilizado.
5. Para cada uma das possíveis chamadas dos métodos *getPID*, *getStatus*, *getCPU* e *getMem*, existe um processo e um estado associado. Por exemplo, quando o método

getPID é chamado, ele executa o processo *getPID*. Enquanto isso, ele permanece no estado GETPID.

Deve-se notar que a fase de análise fica facilitada devido, principalmente, ao encapsulamento provido pelas classes contidas na biblioteca SNMPLib. Realmente, pode-se observar que os diagramas expressam muito pouco sobre conceitos de gerência de redes.

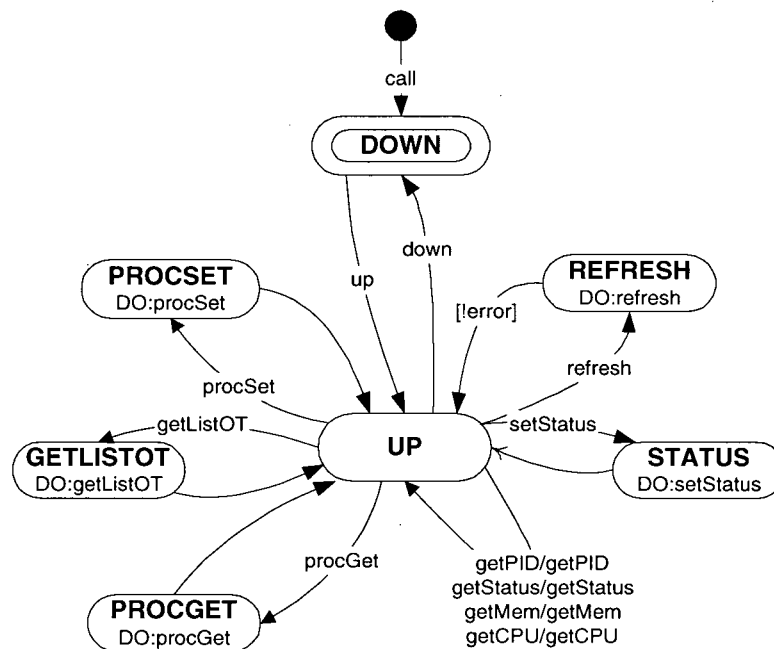


Figura 6-8 Diagrama de Estados da Classe procMO

Mesmo sendo muito simples o exemplo dado, pode-se observar que qualquer crescimento na complexidade da solução do problema irá ocorrer no sentido das peculiaridades inerentes do problema. Realmente, os diagramas aqui apresentados são básicos e podem ser tomados como ponto de partida para problemas mais complexos. Toda extensão feita aos diagramas serão freqüentemente resultados das necessidades particulares envolvidas na solução de um problema.

- **Fase de Projeto**

A implementação, particularmente em C++, pode ser feita diretamente, devido a pouca complexidade do problema. Logicamente, se o sistema analisado fosse mais complexo, uma fase de refinamento mais detalhado poderia ser necessário.

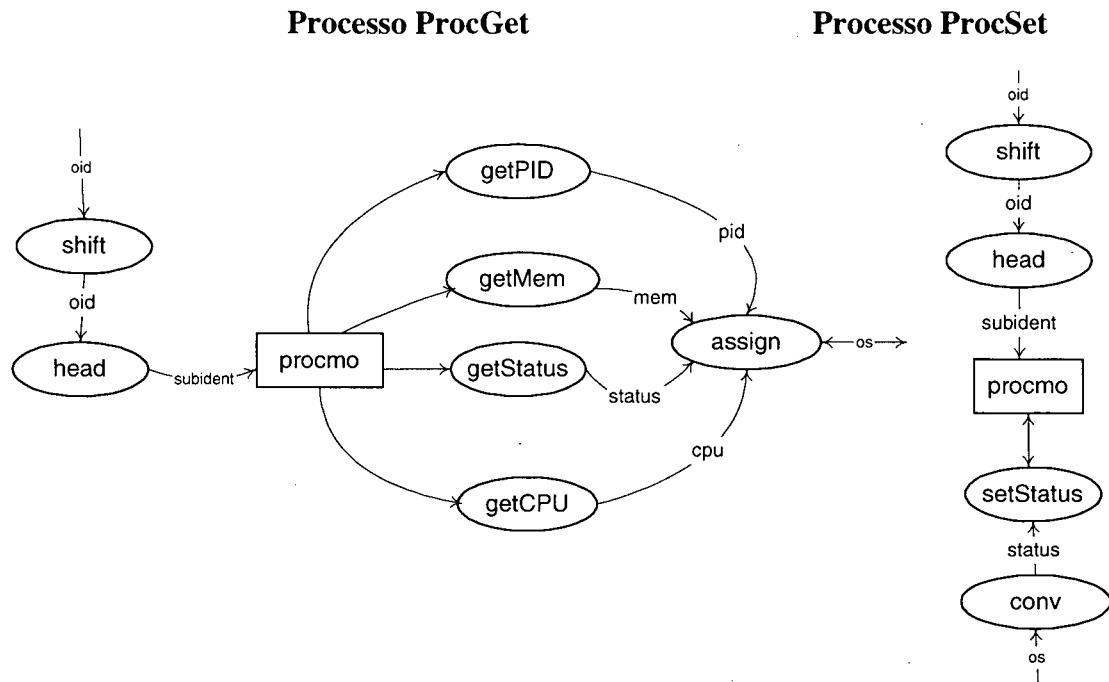


Figura 6-9 Diagrama de Fluxo de Dados para a Classe *procMO*

Como pode ser notado na figura 6.3, as interfaces públicas das classes *process*, *procMO* e *procmond* são relativamente simples. O conteúdo do arquivo *process.h*, mostrado na figura 6.10, é o código da interface pública da classe *process*. A única a respeito deste código é o uso da classe *Istring*, no lugar de um ponteiro para char, visando simplificar o uso de *strings* em C++. Assim sendo, operadores como “+” e “=”, cuja semântica e sintaxe são óbvias, podem ser largamente utilizados.

O arquivo *process.cpp* da figura 6.11, implementa as definições dos métodos da classe *process*. Nota-se nas definições dos métodos, o uso da função *system* que envia um comando para ser executado pelo ambiente de execução de comandos (*shell*, no UNIX). Por exemplo, quando a chamada *system* é executada no método *up*, faz com que seja executado o comando que inicializa o processo monitorado.

Outra função sendo usada, é a chamada de sistema *kill* para enviar uma interrupção a um processo. Quando ela é usada nos métodos *refresh* e *down*, a interrupção -9 é enviada, abortando o processo.

```
#ifndef classProcess
#define classProcess
#include <istring.h> // Declaração da classe IString

class process
{
    private:
        update();
    protected:
        IString command;
        IString path;
        int pid;
        char status;
        int cpu;
        int mem;
    public:
        process(const IString & Comm, const IString & Path);
        int update();
        int refresh();
        int down();
        int getPID() { return pid; }
        char getSTATUS() { return status ;}
        int getCPU() { return cpu; }
        int getMem() { return mem; }
        void setStatus(char s) { status = s; }
        void setCPU(int p) { cpu = p; }
        void setMem(int p) { mem = p; }
};
#endif
```

Figura 6-10 Arquivo *process.h*

```
#include <stdlib.h>
#include <signal.h>
#include <stdlib.h>
extern "C"
{
    #include <sys/resource.h>
    #include <errno.h>
};
#include <time.h>
#include "process.h"
process::process(const IString& Comm,const IString& Path)
{
    IString cmd = Path + "/" + Comm;
    int ret = system(cmd);
    char c[100];
    int p;
    ret = system("/usr/bin/ps -efo comm,pid > /tmp/psfile1.log");
    ifstream psf("/tmp/psfile1.log",ios::in);
    psf >> c;
    psf >> c;
    do
    {
        psf >> c;
        psf >> p;
    }
    while((!psf.eof()) && (strcmp(c,Comm)));
    if(!strcmp(c,Comm))
    {
        command = comm;
        path = Path;
        pid = p;
        cpu = mem = -1;
    }
    timer = (time_t) 0;
}
int process::refresh()
{
    if(!kill(pid,-1))
    {
        update();
        return 0;
    }
    else
    {
        return -1;
    }
}
int process::up()
{
    IString cmd = path + "/" + command;
    return system(cmd);
}
```

Figura 6-11 Arquivo process.cpp (parte 1)

```
int process::down()
{
    if(!kill(pid,-9))
    {
        return 0;
    }
    else
    {
        return 1;
    }
}

int process::update()
{
    char line[160];
    char cmd[80];
    sprintf(cmd,"ps vg %d > /tmp/ps.log",pid);
    if(!system(cmd))
    {
        ifstream psf("/tmp/ps.log",ios::in);
        psf.getline(line,180); // read header
        psf.getline(line,180);
        float pcpu, pmem;
        sscanf(line,"%*d %*s %*c %*s %*d %*d %*d %*s %*d %*d %f %f %*s",
                &status,&pcpu,&pmem);
    }
    else
    {
        error("update--ps command failed: ps -ef");
    }
    cpu = (int) pcpu;
    pmem = (int) pmem;
    return 0;
}

int process::setStatus(char c)
{
    switch(c)
    {
        case 0:
            down();
            break;
        case 1:
            up();
            break;
        case 2:
            refresh();
            break;
    }
}
```

Figura 6-12 Arquivo *process.cpp* (parte 2)

A figura 6.12 mostra o arquivo *procmo.h* que implementa a interface pública da classe *procMO*. Nota-se neste arquivo, que *procMO* é declarada como derivada das classes *process* e *sMO*. Além disso, os métodos virtuais *procGet*, *procSet* e *getListOT* de *sMO* são sobrepostos.

Pode ser notado que são declarados como privados, quatro atributos da classe *sOS*. atributos são utilizados na construção de atributos da classe *sOT*, declarados como protegidos. Além disso, todos são atributos de classe, pois são declarados como membros estáticos.

```
#ifndef ClassProcMO
#define ClassProcMO
#include <istring.h>
#include "sMO.h"
#include "process.h"
#include "sosint.h"
class procMO : public process, sMO
{
    private:
        static sOSInteger osPID;
        static sOSInteger osMEM;
        static sOSInteger osCPU;
        static sOSOCTECT osSTAT;
    protected :
        static sOT procPID;
        static sOT procMEM;
        static sOT procCPU;
        static sOT procSTAT;
    public:
        int procGet(sOS *os, const sOID& o);
        int procSet(const sOS *os, const sOID& o);
        void getListOT(sTree &tree);
};
#endif
```

Figura 6-13 Arquivo procmo.h

A figura 6.13 mostra o arquivo *procmo.cpp* que contém as definições dos métodos declarados em *procmo.h*. Pode-se perceber que as primeiras linhas inicializam os atributos da classe *sOT*, sendo três deles da classe *sOSInteger* e um, da classe *sOSOCTECT*, classes derivadas de *sOS*.

Em todas as chamadas do construtor da classe *sOT*, o primeiro parâmetro, indicando a sua posição na MIB, é um *string* que começa com "1.3.6.1.4.1.9999". Este *string* é também a raiz da subárvore privada.

Outro fato a ser observado é a chamada do método *copy* de *sOS*, tanto em *procGet* como em *procSet*. Em *procGet*, isto é feito para poder converter um *integer* ou *char* para *sOSInteger* e colocá-lo na área apontada por *os*. Em *procSet*, isto é necessário para converter o valor apontado pelo ponteiro *os* para um *integer* ou *char*.

```
#include "procmo.h"
procPID = sOT("1.3.6.1.4.1.9999.1.1", "procPID", sOT::ReadOnly, procMO::osPID);
procMEM = sOT("1.3.6.1.4.1.9999.1.2", "procMEM", sOT::ReadOnly, procMO::osMEM);
procCPU = sOT("1.3.6.1.4.1.9999.1.3", "procCPU", sOT::ReadOnly, procMO::osCPU);
procSTAT = sOT("1.3.6.1.4.1.9999.1.4", "procSTAT", sOT::ReadWrite, procMO::osSTAT);
void procMO::getListOT(sTree &tree)
{
    tree.add(&procPID);
    tree.add(&procMEM);
    tree.add(&procCPU);
    tree.add(&procSTAT);
}
int procMO::procGet(const sOS *os, const sOID& o)
{
    sOID o1 = o;
    o1.shift(1);
    switch(o1.head())
    {
        case 1:
        {
            sOSInteger oint=getPID();
            oint.copy(os);
            break;
        }
        case 2:
        {
            sOSInteger oint=getMem();
            oint.copy(os);
            break;
        }
        case 3:
        {
            sOSInteger oint=getCPU();
            oint.copy(os);
            break;
        }
    }
}
```

Figura 6-14 Arquivo procmo.cpp (parte1)

```

        case 4:
        {
            sOSOCKET ochar = getStatus();
            oint.copy(os);
            break;
        }
    }
}
int procMO::procSet(const SOS *os, const sOID& o)
{
    sOID o1 = o;
    o1.shift(1);
    switch(o1.head())
    {
        case 4:
        {
            sOSInteger oint;
            os->copy(&oint);
            int i = oint;
            if(setStatus( i )
            {
                warn("procSet -- error in procedure set");
                ret = -1;
            }
        }
    }
}

```

Figura 6-15 Arquivo procmo.cpp (parte 2)

Observe que na instrução *switch* do método *procSet* somente existe um *case*. Isto acontece porque unicamente o atributo de que indica o *status* do objeto gerenciável pode ser alterado

A figura 6.14 mostra o arquivo *procmond.h*, cujo conteúdo é a declaração da classe *procMonD*. Pode-se notar que a classe *procMonD* também tem um atributo de classe, o membro *rootTree*. De fato, isto implica que todas as instâncias de *procMonD* compartilham a mesma raiz da subárvore privada.

```
#ifndef ClassProcMonD
#define ClassProcMonD
#include "ssagent.h"
#include "soid.h"
#include "procmo.h"

class procMonD : public sSagent
{
protected:
    static sOID rootTree;
    procMO proc;
public:
    procMonD ();
    int run();
    int procGet(sOS *os,sOID oid);
    int procSet(sOS *os, sOID oid);
};
#endif
```

Figura 6-16 Arquivo *procmond.h*

A figura 6.15 mostra o arquivo *procmond.cpp*, trazendo a implementação dos métodos declarados no arquivo *procmond.h*. Na figura 6.14, pode-se observar que no construtor são informados quatro parâmetros:

- a raiz da subárvore privada,
- o nome,
- a *password* e
- uma descrição textual do subagente.

Todos estes parâmetros devem ser colocados no arquivo de configuração do agente SNMP, na maioria dos sistemas UNIX, o arquivo “/etc/snmpd.conf”.

Deve ser notado no método *run*, que *open* é chamado com prioridade de -1 para o registro de árvore privada. Além disso, a operação informada sobre a interface SMUX é de RW (leitura e escrita).

```

#include "procmond.h"
#include "soid.h"
#include "sot.h"
rootTree = sOID("1.3.6.1.4.1.9999"); // raiz da subarvore
procMonD::procMonD():sSagente(rootTree,"proc","proc_password","procmond daemon") { }
int procMonD l::run()
{
    sTree privateTree;
    proc.getListOT(privateTree);
    open(rootTree,privateTree,-1,sSMUX::RW);
    int ret ;
    if(ret = dispatch()) warn("Dispatch -- retorno invalido");
    close();
    return ret;
}
int procMonD::procGet(sOS * os,sOID oid)
{
    sOID oid1 = oid - rootTree;
    int ret = 0;
    switch(oid1.head())
    {
        case 1:
        {
            proc.procGet(os,oid);
            break;
        }
        default:
        {
            warn("procGet - invalid object type");
            ret = 1;
        }
    }
    return ret;
}
int procMonD::procSet(sOS *os,sOID oid)
{
    int ret = 0;
    sOID oid1 = oid - rootTree;
    switch(oid1.head())
    {
        case 1:
        {
            proc.procSet(os,oid);
        }
        default :
        {
            warn("procSet -- object id inexpected");
            ret = -2;
        }
    }
    return ret;
}

```

Figura 6-17 Arquivo *procmond.cpp*

- **Definição SMI**

A MIB estendida é formada pela MIB-II, controlada pelo agente SNMP, mais a subárvore privada que contém os objetos controlados pelo subagente e que não estão na MIB-II.

A figura 6.16 exibe a subárvore privada (ramo com nomes em itálicos) da MIB controlada pelo subagente *procmond*. Verifica-se que a subárvore foi definida no nó *private*, onde são colocadas as definições de MIB privadas, principalmente de fornecedores.

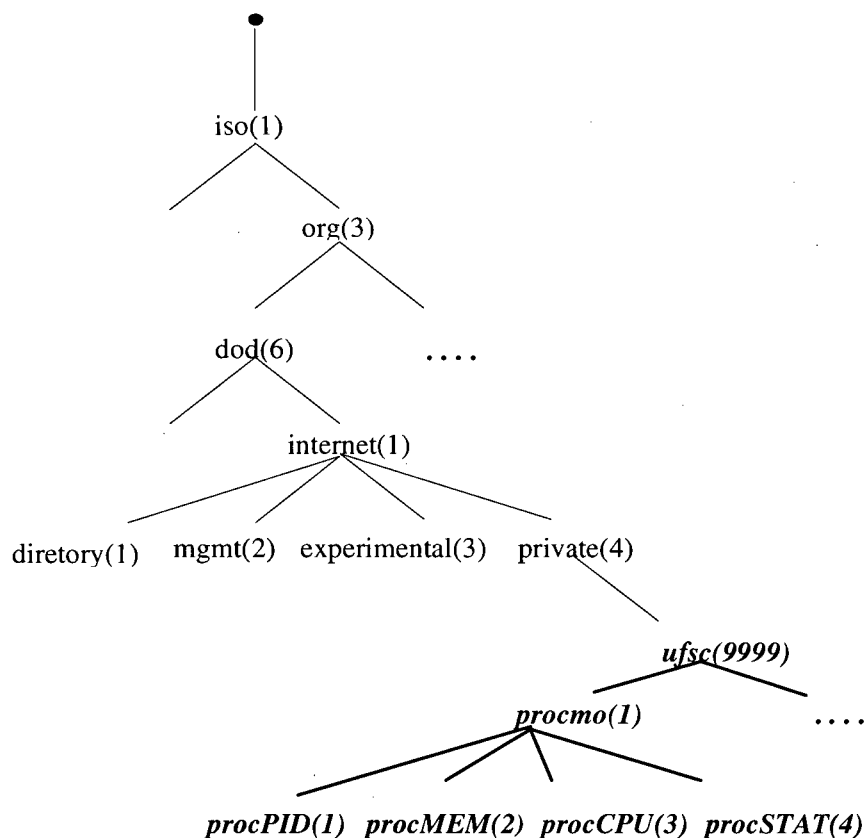


Figura 6-18 Representação da Subárvore Privada Controlada por *procmond*

6.5 Neurolib

Na fase de análise, questões envolvidas com o padrão Internet como protocolo de gerência SNMP, identificação de instância de objetos ou sua estrutura na SMI são deixados para a fase de projeto. O modelo deverá somente incluir informações relevantes ao domínio do

problema, os quais são compreensíveis por alguém não especialista em gerência de redes ou redes neurais.

Na análise, pode-se identificar que a adaptação é uma característica herdada por um objeto gerenciável derivado de uma classe de redes neurais, ou pode ter uma relação de agregação entre uma classe de redes neurais e um objeto gerenciável ou ainda, pode ser uma associação entre classes de objetos gerenciáveis.

6.5.1 Objeto Gerenciável Derivado de uma Rede Neural

- **Análise**

Suponha uma interface que interliga uma estação a uma rede de computadores IEEE 802.3 [34] [35] [36] . Alguns dos atributos desta interface são: pacotes enviados e recebidos, colisões máximas e pacotes enviados com erros. Cada um destes atributos servem para verificar como está o estado da interface, ajudam a detectar algum problema no equipamento que liga a interface ao cabo de rede e a prever possíveis problemas.

Portanto, os atributos da Interface IEEE 802.3 serão tomados para formar o vetor de entrada para uma rede neural. Neste caso, o objeto gerenciável pode ser implementado como uma instância de uma classe derivada de uma rede neural. A figura 6.17 descreve o diagrama de classes desta análise.

- **Projeto**

Em C++, a interface pública pode ser realizada pelo arquivo *ieee8023.h* mostrado na figura 6.18. A implementação da classe pode ser realizada pelo código da figura 6.19, mostrando o arquivo *ieee8023.cpp*.

Na figura 6.20 é mostrada uma aplicação desta classe. A interface de comunicação com o programa é baseada em três arquivos:

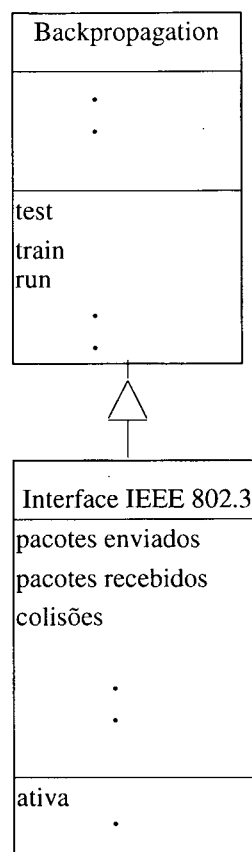


Figura 6-19 Classe Interface *IEEE 802.3*

1. **Arquivo de definições** - contém as definições da rede e tem o formato da figura 6.21. O conteúdo do arquivo é um conjunto de *strings* `<palavra-chave=valor>`, onde *palavra-chave* pode ser:

INPUTS: tamanho da camada de entrada.

OUTPUTS: tamanho da camada de saída.

RATE: taxa de aprendizagem.

MOMENTUM: valor do *momentum* na mudança de pesos.

TOLERANCE: valor da tolerância dentro da qual uma saída será aceita como válida.

No programa da figura 6.20, é usada a instância *def* da classe *fKeywparm* para ler do arquivo de definições, o valores dos parâmetros. A declaração de instância *def* é realizada informando-se no construtor de *fKewyparm*, para cada parâmetro da rede, a palavra-chave e tipo de valor (*integer* para *int*, *string* para *char**, *vector* para *mVec* e *vectorpair* para *mVecPair*). O arquivo de definições no programa da figura 6.20 tem o nome de "ieee8023.def" e pode ser exemplificado pelo arquivo da figura 6.21.

2. **Arquivo de log**: este arquivo, cujo nome é “ieee8023.log”, armazena os diagnósticos a cada *LOOPTIME* segundos efetuados pelo programa.
3. **Arquivo de Treinamento** - cada linha deste arquivo contém um par de vetores: vetor de entrada que contém os valores dos atributos da interface e o vetor de saída com o diagnóstico. Todos os valores de entrada são classificados em três classes:

Valor Baixo (0)

Valor Alto (1)

Valor Alto (2)

Para os valores de saída, tem-se os valores possíveis definidos pela quadro 6.2.

Elemento	Problema
0	Nenhum problema
1	Problema na interface
2	Problema de Cabeamento
3	Problema com <i>driver</i> da placa ou no cabeamento
4	Problema no MAU (<i>transceiver</i>)
5	Problemas no MAU (<i>transceiver</i>) e na Interface
6	Problemas no MAU (<i>transceiver</i>) e no Cabeamento
7	Problemas no MAU (<i>transceiver</i>), no Cabeamento e na Interface

Quadro 6-2 Códigos de Diagnósticos

Observando a figura 6.20, tem-se que o primeiro passo é ler as definições da rede contidas no arquivo de definições. O segundo passo consiste em se obter as definições da rede e armazenar em variáveis que serão usadas na chamada do construtor da classe *IEEE8023*. Este passo é feito através de uma seqüência de chamadas do método *get* da classe *fKewyparm*.

Depois de construída no passo 3, a instância *ieee8023* é carregada no passo 4 e no passo 5, é verificada se a rede foi treinada. Se a rede não foi treinada, no passo seguinte o conjunto de treinamento é lido para ser utilizado no treinamento de *ieee8023* no passo 7. A seguir, novamente é verificada se a *ieee8023* conseguiu ser treinada. Se foi, *ieee8023* está pronta para diagnosticar a rede. Neste caso, a instância entra em um laço infinito. Neste laço, a instância fica aguardando por

LOOPTIME segundos. Os atributos da interface são atualizados e recuperados no vetor *in* no passo 10 para em seguida, o método *recall* produzir um diagnóstico contido no vetor *out*. O vetor *out* é então registrado no arquivo de *log*, para ser analisado pelo gerente da rede.

```
#include "nbp.h"
#include "mvec.h"
class IEEE802_3 : public nBP
{
    private:
        int packetsIn;
        int packetsOut;
        int colisions;
    public:
        void enable();
        void disable();
        IEEE802_3 & operator>>( IEEE802_3& ieee8023, mVec& in);
}
```

Figura 6-20 Arquivo **ieee8023.h**

```
void IEEE802_3::enable()
{
    system("/etc/ifconfig en0 up");
}
void IEEE802_3::disable()
{
    system("/etc/ifconfig en0 down");
}
IEEE802_3 & IEEE802_3::operator>>( IEEE802_3 &ieee8023, vec& in)
{
    .
    .
    // utiliza o comando netstat -i en0 -- código omitido por razoes de brevidade
}
```

Figura 6-21 Arquivo **ieee8023.cpp**

```

#include "fkparm.h"
#include "iee8023.h"
#include<stdlib.h>
#define MAXITER 200000 // loop maximo de 200 000 iteracoes
#define TIMELOOP 600 // Tempo de loop e 600 s = 10 min
int main()
{
    fKeywparm def("INPUTS",integer,"OUTPUTS",integer,"RATE",real,"HIDDEN",integer,
                 "MOMENTUM",real,"TOLERANCE",real,"MINVECS",integer,
                 "MAXVECS",integer);
    def.read("iee8023.defs"); // Passo 1 : Le as definicoes de rede
    /* Passo 2 : obtem as definicoes da rede contidas em def */
    def.get(inputs,"INPUTS");
    def.get(outputs,"OUTPUTS");
    def.get(learnrate,"RATE");
    def.get(hidden,"HIDDEN");
    def.get(momentum,"MOMENTUM");
    def.get(tol,"TOLERANCE");
    int min,max;
    def.get(min,"MINVECS");
    def.get(max,"MAXVECS");
    vecpair minVecPair(ninputs,vn,min),max VecPair(mrow*mcol,vn,min);
    /* Passo 3: cria uma instancia de IEEE8023 */
    IEEE8023 iee8023(inputs,outputs,hidden,tol,learnrate,momentum, minVecPair,max VecPair);
    iee8023.load("iee8023.sav"); // Passo 4: carrega a rede
    if(!iee8023.isTrain()) // Passo 5: Verifica se a rede esta treinada
    {
        mVecPairList vplist;
        ifstream s("iee8023.trn", ios::in);
        s >> vplist; // Passo 6: le os pares de treinamento
        iee8023.train(vplist,MAXITER); // Passo 7: treinamento da rede
        iee8023.save("iee8023.sav");
    }

    if(bp.isTrain()) // Passo 8: Verifica se a rede esta treinada
    {
        ifstream s("iee8023.out",ios::app);
        while(1)
        {
            sleep(LOOPTIME); // o programa fica dormindo por LOOPTIME segundos
            iee8023 >> in; // Passo 9 : le os atributos atuais da interface
            iee8023.recall(in,out); // Passo 10 : diagnostica a interface
            s << out;
        }
    }
    else
    {
        ceer << "Rede nao treinada\n";
    }
}

```

Figura 6-22 Arquivo main_iee8023.cpp

```
INPUTS=5
OUTPUTS=3
HIDDEN=3
TOLERANCE=0.05
MOMENTUM=0.85
MINVECS=
MAXVECS=
```

Figura 6-23 Arquivo iee8023.def

6.5.2 Objeto Gerenciável tendo uma Relação de Agregação com a Rede Neural

Analisando o caso acima, pode-se ainda implementar o objeto gerenciável como tendo uma relação de agregação com uma rede neural. A figura 6.22 ilustra esta forma alternada.

Em certos casos, esta pode ser uma forma melhor. Por exemplo, nem sempre pode ser interessante disponibilizar operações das classes de redes neurais. A operação de treinamento ou os atributos do tamanho da camada oculta, não tem muito significado em um ambiente de rede de computadores. Na implementação, a rede neural pode passar a ser um atributo da Interface IEEE 802.3, inclusive protegida, ou seja, não visível na interface pública da classe Interface IEEE 802.3.

6.5.3 Adaptação como uma Associação entre Objetos Gerenciáveis

Este é o caso em que uma rede neural pode ser modelada como uma associação entre um conjunto de classes de objetos gerenciáveis.

Uma ligação desta associação pode ser representada como uma tupla $\langle i_1, i_2, i_3, \dots, i_n \rangle$ onde i_j é um objeto da classe j que participa da associação.

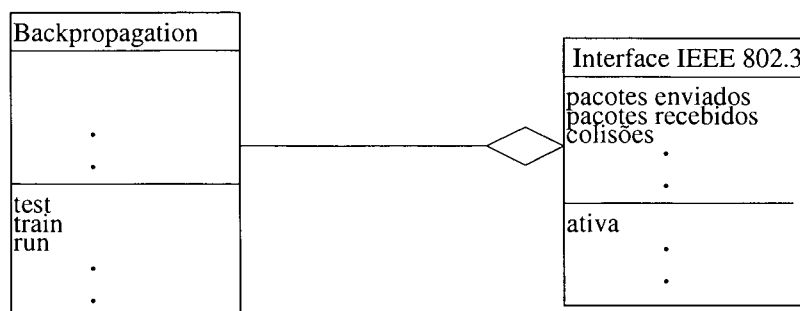


Figura 6-24 Classe Interface IEEE 802.3

Se a associação for implementada como uma classe, então o treinamento e execução são operações desta. Como exemplo, suponha um computador que tenha três interfaces de rede, uma do tipo IEEE 802.3, outra do tipo IEEE 802.5 e outra do tipo FDDI. Todas estas interfaces ligando as três redes IP diferentes com acesso a Internet. A questão é escolher qual interface será a rota *default* [06], ou seja, por qual interface serão enviados os pacotes IP destinados a comunidade Internet. Independente de ser um problema solucionável ou não com redes neurais, mas pensando-se unicamente como exemplo de associação, pode-se formar uma rede neural como uma associação entre classes :

- Interface IEEE 802.3
- Interface IEEE 802.5
- Interface FDDI
- Camada IP

Portanto, uma vez treinada, a rede neural representa a associação formada por um conjunto de ligações <interfIEEE8023,interfIE8025,interfFDDI,IP> onde interfIEEE8023 representa uma instância da classe Interface IEEE 802.3, interfIE8025 uma instância da Classe Interface IEEE 802.5, interfFDDI uma instância da classe Interface FDDI e IP uma instância da classe Camada IP. Pode-se pensar que cada uma destas instâncias representa um estado de um determinado objeto. Portanto, a ligação acima, a instância da camada IP pode ter uma rota default apontando, por exemplo, para a Interface IEEE 802.3.

7. BIBLIOTECA DE CLASSES DE GERÊNCIA DE REDES - SNMPLIB

A Biblioteca de Classes de Gerência de Redes, SNMPLib, é uma coleção de classes que permite a construção de *softwares* aplicados à gerência de redes de computadores.

O objetivo principal da biblioteca é prover o suporte necessário para estender a funcionalidade de agentes SNMP, procurando encapsular nas classes, o código para realizar tarefas de baixa abstração como negociação de conexão, registro de MIBs privadas, construção, recepção e envio de PDUs.

As classes definidas procuram abstrair no modelo Internet, a idéia de um subagente. O subagente é uma forma de estender a funcionalidade de um agente, fazendo-o controlar outros objetos gerenciáveis para os quais não foi projetado.

A SNMPLib contém classes que modelam objetos gerenciáveis, PDUs SNMP, MIB, interfaces e subagentes SMUX. A construção de um subagente pode ser feita utilizando-se destas classes básicas, sendo que em muitas vezes, basta construí-lo através de uma nova classe derivada da classe *sAgent*.

7.1 Diagrama de Classes

O diagrama de classes OMT da SNMPLib é aquele apresentado na figura 7.1. Nota-se que neste diagrama estão anotados aspectos gerais das classes, permitindo-se verificar, por exemplo, que existem poucas relações de herança, mas as classes possuem muitas relações de associação e agregação.

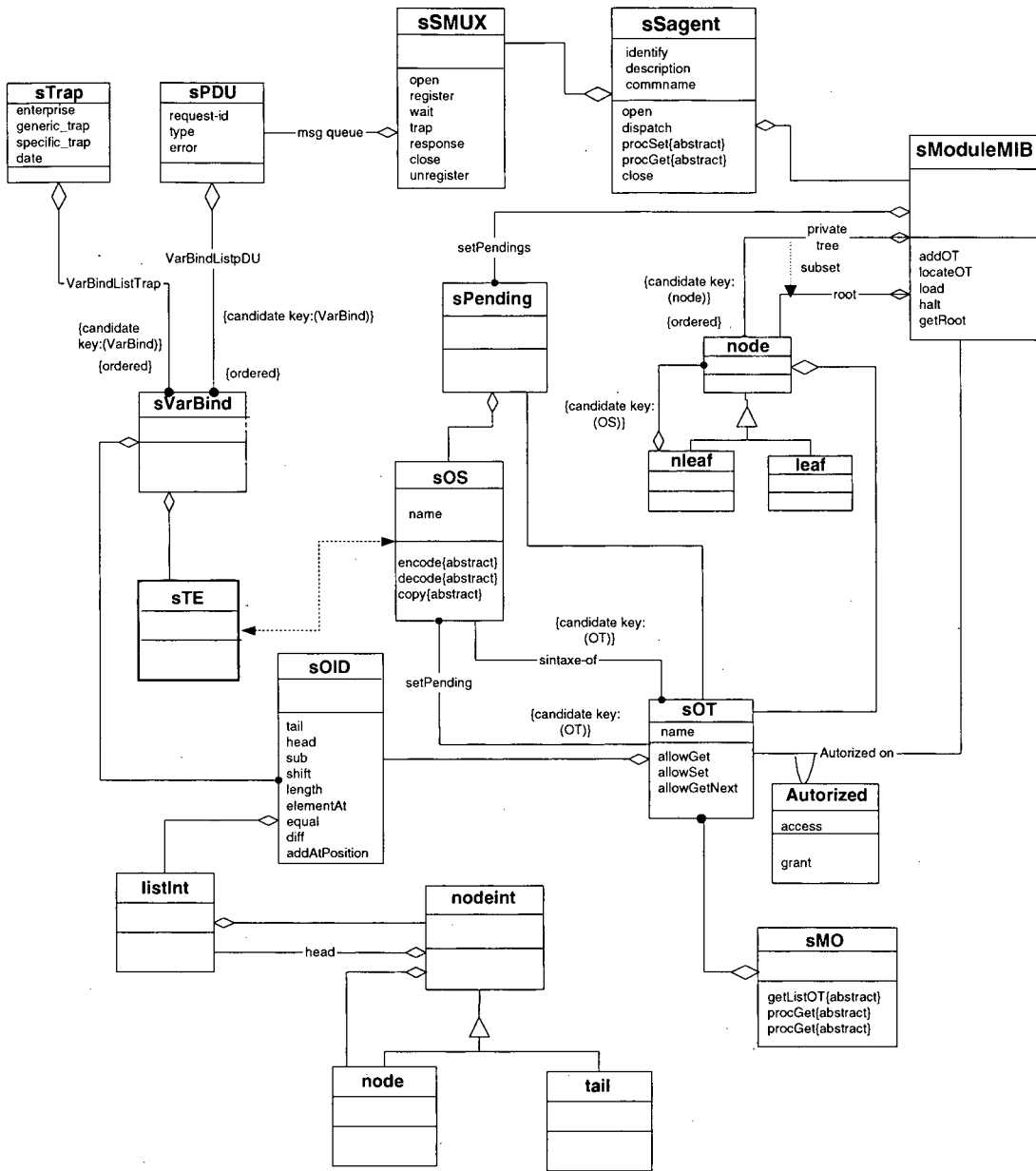


Figura 7-1 Diagrama de Classes de Gerência de Redes

7.2 sSagent

A classe *sSagent* é a classe principal da Biblioteca de Gerência de Redes, a qual implementa um subagente SMUX. Como será visto, à partir de uma instância de *sSagent*, as demais instâncias das outras classes são criadas e removidas. Portanto, a classe *sSagent* mantém um relacionamento de agregação direta ou indiretamente com todas as demais classes.

• Descrição

Uma instância de *sAgent*, descrita pela figura 7.2, compõe-se de três partes principais: um objeto interface SMUX, um objeto Módulo MIB contendo vários tipos de objetos cadastrados e uma operação *dispatch*.

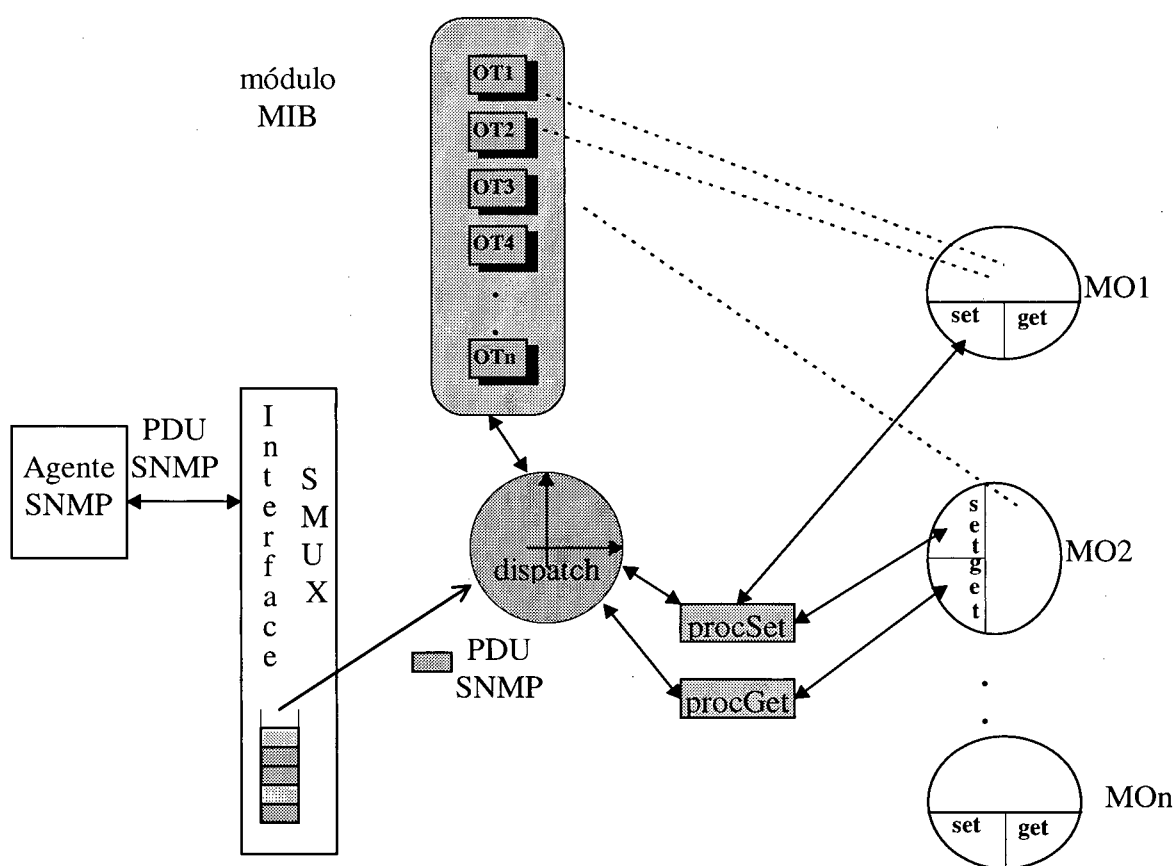


Figura 7-2 Descrição das Partes de um Subagente

Cada tipo de objeto é a representação na MIB de uma característica de um objeto gerenciável. Esta característica pode ser implementada como um atributo ou um método na classe *sMO*.

A principal operação do subagente é de *dispatch*. Através da interface SMUX, a operação *dispatch* obtém uma mensagem SNMP, contendo uma operação *Get*, *Get-Next* (tratada como uma operação *Get*) ou *Set*, para ser atuada sobre os objetos gerenciáveis controlados pelo subagente.

A interface SMUX serve para a comunicação entre o subagente e o agente SNMP. A interface fornece serviços como recepção e envio de PDUs, envio de *traps* e registro de tipos de objetos.

- *dispatch*

Como pode ser visto nos diagramas de estado, na figura 7.3, e de fluxo de dados, na figura 7.4, a operação *dispatch* permanece em um laço infinito, saindo somente quando há uma mensagem *closeRequest*.

Cada vez que entra no estado WAITING, o *dispatch* insere um pedido por recepção de uma PDU à interface SMUX. Se não há alguma PDU na fila de recepção, o *dispatch* fica aguardando até que chegue uma. Se aparecer uma PDU na fila de recepção, o *dispatch* verifica o tipo da mensagem, representada na figura 7-3 pelo estado ROUTING, efetuando a operação *getType*.

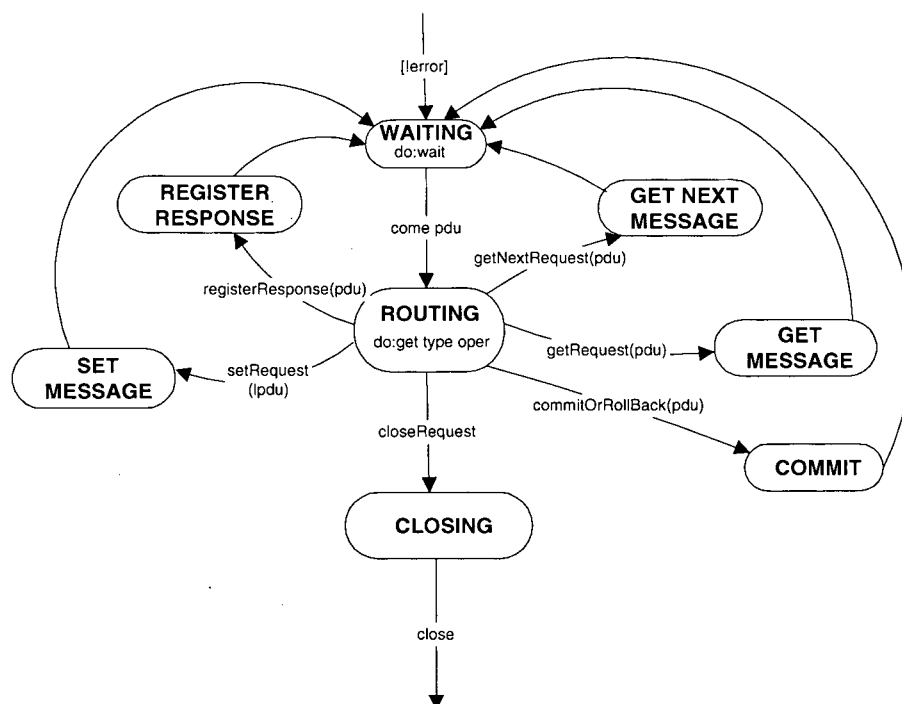


Figura 7-3 Estado DISPATCH do Subagente

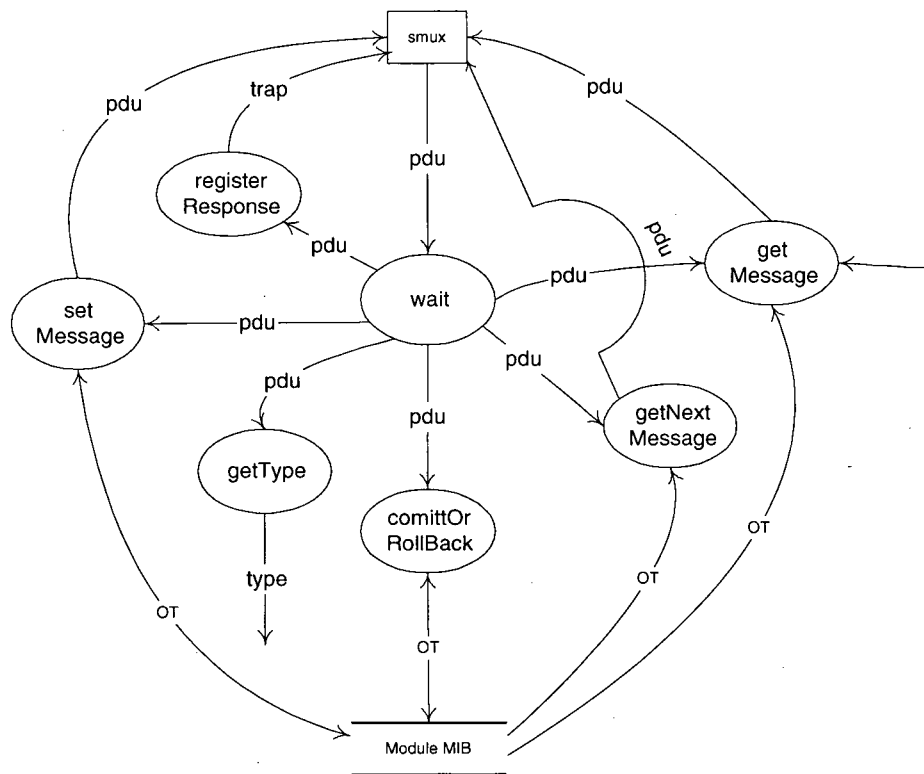


Figura 7-4 Processo Dispatch

Dependendo do tipo, a mensagem pode ir para um dos processos do subagente:

1. **RegisterResponse** de um registro de subárvore: o *dispatch* chama a operação *registerResponse*, que verifica se a PDU é uma resposta de erro. Se for, *registerResponse* retorna um erro e o subagente interrompe o processamento. Se não for uma resposta de erro, *registerResponse* envia um *trap* e retorna sem erro.
2. **GetRequest**: *dispatch* envia a PDU para o processo *getMessage*, entrando no estado GETMESSAGE, cujo diagrama de estados como pode ser visto na figura 7.5. *getMessage* efetua o desencapsulamento e verificação da corretude das informações contidas na PDU. Para cada identificador de objeto na lista de variáveis (instância da *varBindList*) contidas na PDU, *getMessage* resumidamente:
 - Obtém o identificador de objeto no estado 8. Se houver algum erro, *noSuchName* é retornado e o estado 22 é alcançado, encerrando o laço.

- No estado 9, localiza o tipo de objeto correspondente.
- Verifica se o subagente tem permissão de acesso ao tipo de objeto para realizar a operação *get*. Esta situação é representada pelo estado 10, chamando a atividade *allowGet*. Se não tem, *noSuchName* é retornado e o estado 22 é alcançado, encerrando o laço.
- Se tiver permissão, obtém e copia a sintaxe correspondente do tipo de objeto nos estados 11 e 12.
- Chama operação *procGet* do subagente no estado 13.
- Processa o próximo tipo de objeto na lista de variáveis da PDU.

Depois do estado 22, final do laço, uma PDU de resposta contendo os identificadores de objetos e os valores respectivos é enviada para o agente. A PDU é repassada pelo agente ao gerente se tudo estiver correto. Esta situação é representada pelos estados 5 e 6.

3. ***Get-NextRequest***: *dispatch* envia a PDU para o processo *getNextMessage*. *getNextMessage* é um processo semelhante a *getMessage*, com uma exceção: ele atua no tipo de objeto com o identificador lexicograficamente mais próximo daquele informado.
4. ***SetRequest***: Da mesma forma que *getMessage*, *setMessage*, efetua o desencapsulamento e verificação da correteza das informações da PDU, obtendo os identificadores e os tipos de objetos associados no módulo MIB. As diferenças com relação ao método *getMessage* são que *setMessage*:
 - Verifica se o subagente tem permissão adequada de acesso ao tipo de objeto para realizar a operação *Set*, ou seja, permissão de escrita.
 - Se tiver permissão, *setMessage* coloca o tipo de objeto, com seus respectivos valores retirados da PDU, em uma lista de pendências para ser processada mais tarde.
5. ***CommitOrRollBack***: o *dispatch* envia a PDU para o processo *CommitOrRollBack*. Se a PDU contiver um subtipo de mensagem *commit*, então todos os pedidos por operações

Set da lista de pendências são efetuados. Caso contrário, se for um subtipo de *rollBack*, então os pedidos são cancelados.

Desta forma, a operação *dispatch* pode chamar a operação *procGet* para uma PDU *Get* ou *Get-next*, ou ainda, chamar a operação *procSet*, se for uma PDU *set*. Visto que a classe *sAgent* é abstrata, estes métodos devem ter implementações fornecidas pelo usuário das classes derivadas.

- Métodos *procGet* e *procSet*

Os métodos *procGet* e *procSet* são métodos abstratos no subagente, ou seja, as classes derivadas de *sAgent* devem fornecer uma implementação para cada um deles.

Tanto no método *procSet*, como no método *procGet*, existem duas possibilidades para o tratamento da operação:

- Se o tipo de objeto não for um tipo construído, então é chamado diretamente um método da instância de uma classes derivada *desMO* que tem acesso àquele atributo.
- Se o tipo de objeto for um tipo construído, então é chamado o método *procGet* ou *procSet*, correspondente à instância de uma classe derivada de *sMO* que implementa o objeto gerenciável correspondente. Estes dois métodos nesta classe derivada devem ser fornecidos pelo usuário, ou seja, eles são métodos abstratos.

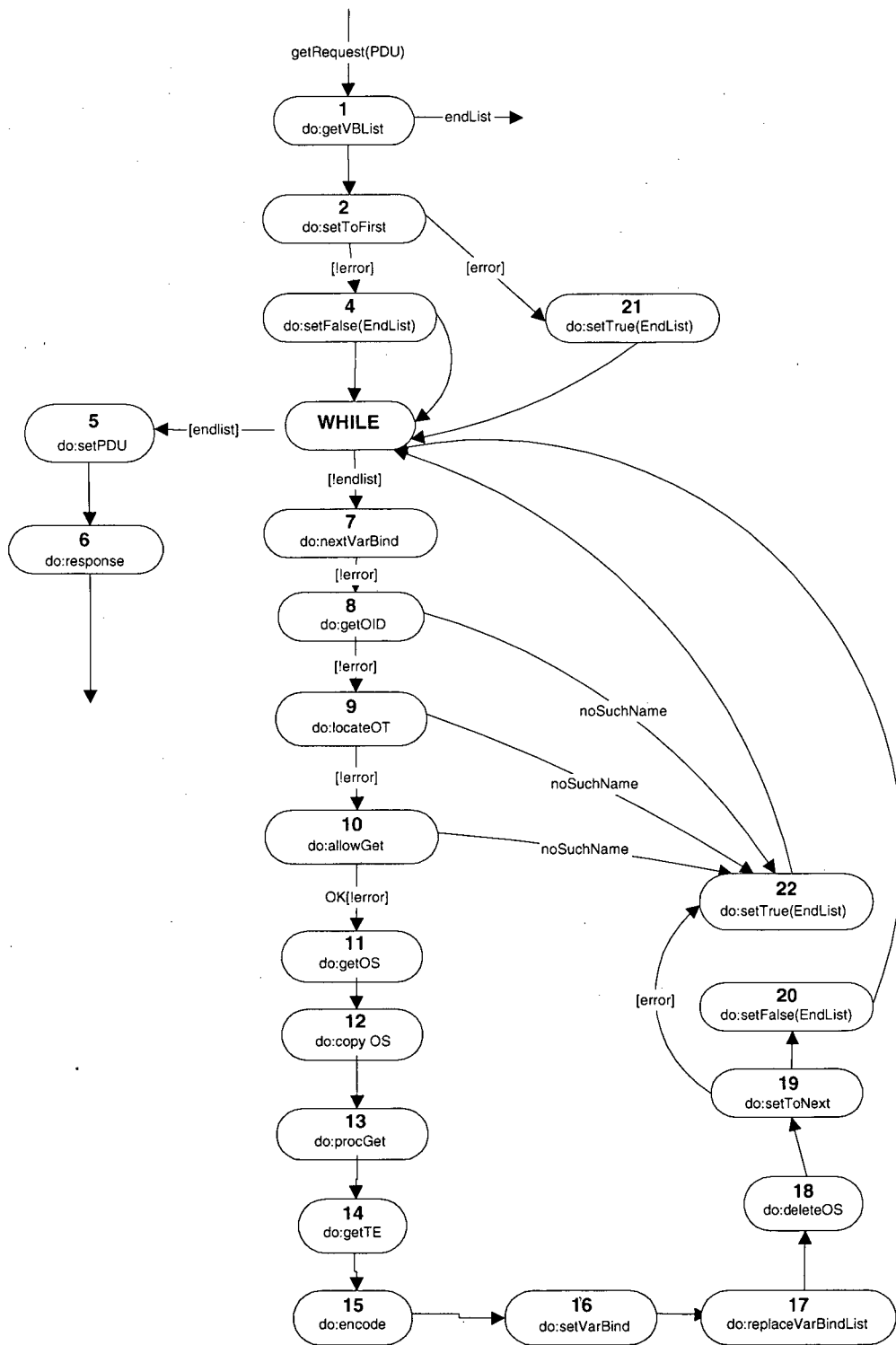


Figura 7-5 Estado GETMESSAGE

A figura 7.6 resume como deve ser o código do método *procGet*, na linguagem C++. Como pode ser notado, o método *procGet* recebe como parâmetros um ponteiro para uma instância de OS e uma instância de OID contido na *PDUGet*.

O primeiro passo consiste na remoção da parte referente à raiz (objeto *root*) do módulo da MIB privada, através do método *diff* aplicado ao objeto *o*. Neste ponto, o primeiro elemento de *o*, identifica à qual atributo se destina a mensagem. No segundo passo, o método *head* aplicado a *o*, obtém este valor.

```
void SA::procGet(OS* s,sOID o)
{
    .
    .
    o.diff(root);          // Passo 1 - retira de o, a parte da raiz da MIB privada
    // o tem somente subidentificadores da MIB privada que este subagente manipula
    switch(o.head( ))      // Passo 2 - Obtem a identificação do atributo
    {
        case 1:           // Passo 3 - Atributo não construido
        {
            .
            .
            sOSInteger s1 = mo1.getAtrib1(); //Passo 4 - Obtém o valor do atributo
            s1.copy(*s);          // Passo 5 - Copia para s, o valor de s1
            break;
        }
        case 2:           // Passo 6 - Segundo Atributo - Atributo SMI não construido
        {
            .
            .
            sOSOCTECT s2 = mo2.getAtrib2(); // Passo 7 - Obtém o valor do atributo
            s2.copy(*s);          // Passo 8 - Copia para s, o valor de s2
            break;
        }
        case m:           // Passo 9 - Atributo m - Atributo SMI construido
            mom.procGet(s,o);    //Passo 10 - Delega para o objeto mom resolver a operação
            break;
            .
            .
        }
    }
}
```

Figura 7-6 Método *procGet* de uma Instância SA da Classe *sAgent*

Em seguida, a instrução *switch* desvia o controle para a instrução *case* correspondente ao atributo encontrado. Se o atributo encontrado for não construído, então é realizado um procedimento, implementado pelo usuário, para resolver a operação.

No terceiro passo, na afirmação *case* para o primeiro atributo, é efetuado uma chamada para um método do objeto gerenciável retornando um valor inteiro. Como é definido um operador de conversão de um valor *int* para *sOSInteger*, no fim deste passo, se tem em *s2*, o valor do primeiro atributo na sintaxe *sOSInteger*.

Se o atributo for um atributo construído, então é delegado ao método *procGet* do objeto gerenciável correspondente para resolver a operação SNMP *Get* ou *Set*.

De uma forma semelhante, podemos resolver um pedido de uma operação SNMP *Set*. A figura 7.7 mostra o método que é aplicado pelo subagente no caso de uma operação *Set*.

Diferentemente de *procGet*, em *procSet*, *s* aponta para um área que contém o valor a ser colocado no atributo do objeto gerenciável. Então, antes de aplicar um método para mudar o atributo, *s* é copiado para uma instância da classe *sOS* correspondente ao tipo de atributo (na figura 7.7, passos 4 e 5).

- Métodos *open* e *close*

Um vez criado um objeto da classe *sSagent*, para que o método *dispatch* possa ser chamado, o método *open* deve ser efetuado primeiro.

O método *open* realiza as seguintes funções:

1. Cadastra, no módulo MIB, todos os tipos de objetos manipulados pelo subagente. Como o *sSagent* implementa diretamente somente os tipos de objetos não construídos, o subagente chama o método *getListOT* em cada instância de MO que modela um tipo construído. A classe *Tree* é usada para agrupar todos os tipos de objetos e registrá-los numa única operação, através do método *open*. O método *registerOT* da classe *Module MIB* pode ser usada para registros individuais de tipos de objetos.
2. Estabelece uma interface SMUX entre o agente e o subagente - As características do protocolo SMUX obriga ao subagente a fornecer o nome da raiz, a prioridade na manipulação e o modo de acesso da subárvore para o estabelecimento de uma conexão.

A figura 7.8 resume como seria um método *run*, de uma classe derivada da classe *sSagent*, que efetua o método *open* e depois chama o método *dispatch*.

```

void SA::procSet(OS* s,SOID o)
{
    .
    .
    o.diff(root);          // Passo 1 - retira de o, a parte da raiz da MIB privada
    // o contem somente subidentificadores da MIB privada que este subagente manipula
    switch(o.head( ))     // Passo 2 - Obtem a identificação do atributo
    {
        case 1:           // Passo 3 - Atributo não construido
        {
            .
            sOSInteger s1;
            s1.copy(*s);          // Passo 4 - Cópia para s1, o valor de s
            mo1.setAtrib1((int) s1); // Passo 5 - Obtém o valor do atributo
            s1.copy(*s);
            break;
        }
        case 2:           // Passo 6 - Segundo Atributo - Atributo não construido
        {
            .
            sOSOCTECT s1;
            s1.copy(*s);          // Passo 7 - Cópia para s1, o valor de s
            mo1.setAtrib2((int) s1); // Passo 8 - Obtém o valor do atributo
            s1.copy(*s);
            break;
        }
        case m:           // Passo 9 - Atributo m - Atributo SMI construido
        {
            .
            mom.procSet(s,o);    //Passo 10 - Delega para o objeto mo m resolver a operação
            break;
            .
        }
    }
}

```

Figura 7-7 Método *procSet* do Subagente

Os cinco primeiros passos, estão associados à construção da raiz *rootTree*, os atributos de objetos *tacPID* e *tacStatus*, do objeto gerenciável *tableMO* e da subárvore *privateTree*. Em seguida, *tacPID* e *tacStatus* são adicionados ao objeto *privateTree* nos passos 6 e 7. No passo 8, ao objeto *privateTree* é adicionado (operador “+” aplicado a *privateTree*) a lista de tipos de objetos obtidos de *tacMO* (método *getListOT*).

Como a *privateTree* contém todos tipos de objetos manipulados pelo subagente, pode-se agora chamar o método *open* de *sAgent*, fornecendo como parâmetros: o objeto *rootTree* que contém a raiz da subárvore, o objeto *privateTree*, a prioridade -1 e o modo de acesso de leitura e escrita.

Em seguida, o método *dispatch* pode ser chamado. Ele ficará executando indefinidamente enquanto não acontecer, ou um erro no método (informado com um valor de retorno diferente de zero), ou pedido de fechamento de conexão por parte do agente. O último passo realizado, depois que *dispatch* retornar, será a chamada para o método *close*, que fecha todas as estruturas de dados abertas.

```
void SA::run()
{
    sOID rootTree("1.3.6.1.4.1.9999");           //Passo 1 - raiz da subarvore
    sOT tacPID("1.3.6.1.4.1.9999.1","tacPID",sOT::ReadOnly); // Passo 2 - Atributo tacPID
    sOT tacStatus("1.3.6.1.4.1.9999.2","tacStatus",sOT::ReadWrite); // Passo 3 - Atributo tacStatus
    tacTable tableMO;                          // Passo 4 - Objeto de uma Classe
                                              //      derivada de sMO

    .
    .
    Tree privateTree;                          // Passo 5 - Subárvore Privada
    privateTree.add(tacPID);                    // Passo 6 - Adiciona tacPID na árvore
    privateTree.add(tacStatus);                // Passo 7 - Adiciona tacStatus na árvore
    privateTree = privateTree + tableMO.getListOT(); // Passo 8 - concatena privateTree com
                                              // a árvore manipulada pelo objeto tableMO

    // Realiza a abertura de uma conexão SMUX com o agente e o registro dos Tipos
    // de Objetos contidos em privateTree no módulo MIB
    open(rootTree,privateTree,-1,sSMUX::RW); // Passo 9

    .
    .
    if(dispatch()) // Passo 10 - Efetua o loop p/ dispatchar o atendimento das pds
    {
        error("Erro executando o método dispatch");
    }

    .
    .
    close(); // Passo 11
}
```

Figura 7-8 Método *run* da Classe Subagente

7.3 sMO

Um objeto gerenciável é uma abstração de um recurso de *hardware* ou *software*. Por exemplo, um processo, uma interface de rede ou uma área em disco são recursos que podem ter características gerenciáveis através de um protocolo de gerência.

Os objetos gerenciáveis são modelados por classes derivadas da classe *sMO*. A classe *sMO* é uma classe abstrata porque ela tem definidos três métodos abstratos: *procSet*, *procGet* e *getListOT*.

Os dois métodos *procGet* e *procSet* tem a mesma função que os métodos correspondentes *procGet* e *procSet* para o *sagente*, ou seja, realizar a operação *Get* ou *Set* em um tipo de objeto definido como atributo da classe *sMO*. O método *getListOT* retorna todos os tipos de objetos de um objeto gerenciável.

```

void SA::procGet(OS* s,sOID o)
{
    .
    .
    o.shift();          // Passo 1 - retira de o, a parte referente ao Identificador deste Objeto Gerenciável
    // o tem somente subidentificadores da MIB privada que este Objeto Gerenciável manipula
    switch(o.head())   // Passo 2 - Obtem a identificação do atributo
    {
        case 1:        // Passo 3 - Atributo não construído
        {
            .
            .
            sOSInteger s1 = getAtrib1();    //Passo 4 - Obtém o valor do atributo
            s1.copy(*s);                    // Passo 5 - Copia para s, o valor de s1
            break;
        }
        case 2:        // Passo 6 - Segundo Atributo - Atributo SMI não construído
        {
            .
            .
            sOSOCTECT s2 = getAtrib2();    // Passo 7 - Obtém o valor do atributo
            s2.copy(*s);                    // Passo 8 - Copia para s, o valor de s2
            break;
        }
        case m:        // Passo 9 - Atributo m - Atributo SMI construído
        {
            .
            .
            mom.procGet(s,o);    //Passo 10 - Delega para o objeto mo m resolver a operação
            break;
        }
    }
}

```

Figura 7-9 Método procGet da Classe Objeto Gerenciável

- Métodos *procGet* e *procSet*

A figura 7.9 descreve como o método *procGet* da classe *sMO* deve ser escrito. Como os passos são os mesmos do método equivalente na classe *sSagent*, se faz desnecessário alguma exposição mais detalhada. Somente, um detalhe deve ser notado: a troca do uso do método *diff*, definido no método *procGet* da classe do *sSagent* pelo uso do método *shift* aplicado ao objeto *o2*

(no começo do algoritmo). Isto acontece pois o método *procGet* da instância *sMO* já recebe o identificador de objeto sem a parte da raiz que foi retirada na classes *Sagent*.

Da mesma forma, o método *procSet* pode ser construído pelo algoritmo resumido pela figura 7.10.

```

void SA::procSet(OS* s,sOID o)
{
    .
    .
    o.diff(root);          // Passo 1 - retira de o, a parte da raiz da MIB privada
    // o tem somente subidentificadores da MIB privada que este subagente manipula
    switch(o.head( ))      // Passo 2 - Obtem a identificação do atributo
    {
        case 1:           // Passo 3 - Atributo não construido
        {
            .
            .
            sOSInteger s1;
            s1.copy(*s);          // Passo 4 - Copia para s1, o valor de s
            mo1.setAtrib1((int) s1); // Passo 5 - Obtém o valor do atributo
            s1.copy(*s);
            break;
        }
        case 2:           // Passo 6 - Segundo Atributo - Atributo SMI não construido
        {
            .
            .
            sOSOCTECT s1;
            s1.copy(*s);          // Passo 7 - Copia para s1, o valor de s
            mo1.setAtrib2((int) s1); // Passo 8 - Obtém o valor do atributo
            s1.copy(*s);
            break;
        }
        case m:           // Passo 9 - Atributo m - Atributo SMI construido
        {
            .
            .
            mom.procSet(s,o);    //Passo 10 - Delega para o objeto mo m resolver a operação
            break;
        }
    }
}

```

Figura 7-10 Método procSet da Classe Objeto Gerenciável

- Método *getListOT*

O método *getListOT* retorna os tipos de objetos que uma instância de *sMO* pode manipular. Ele é definido como um método abstrato, portanto, uma implementação deve ser fornecida pelas classes derivadas de *sMO*.

A figura 7.11 descreve genericamente, como o método *getListOT* pode ser construído. Nota-se que é feita uma chamada para o método *getListOT* dos objetos gerenciáveis contidos no objeto gerenciável em questão.

```
classe unixProc
{
    private:
        OT uPID, uStatus;
        uTable tableMO;          // Objeto de uma Classe derivada da Classe Objeto Gerenciado
    public:
        unixProc()
        {
            uPID.create("1.3.6.1.4.1.9999.1", "tacPID", ReadOnly);
            uStatus.create("1.3.6.1.4.1.9999.2", "tacStatus", ReadWrite);
            uTable.create();
        }
        void getListOT(Tree& t)
        {
            t.add(tacPID);
            t.add(tacStatus);
            t.cat(tableMO.getListOT()); -- concatena t com a árvore manipulada por tableMO
        }
        .
        .
}
```

Figura 7-11 Construtor e método *getListOT* da Classe Objeto Gerenciável

7.4 *sOT*

O termo objeto gerenciável da SMI [20], será aqui redefinido como tipo de objeto. Isto é feito para distinguir instâncias da classe *sMO* que melhor modelam os objetos gerenciáveis que são encontrados na fase de análise. Além disso, é mais adequado usar um objeto gerenciável como definido pela SMI, como um atributo da classe de *sMO*. A classe *sOT* implementa um tipo de objeto, ou seja, um objeto gerenciável da SMI da Internet.

Cada tipo de objeto está associado com uma definição de sintaxe utilizando-se a macro ASN-1 OBJECT-TYPE da SMI [20] [21], mostrada na figura 7.12.

Por exemplo, *sysDescr* é definido como

```
sysDescr OBJECT-TYPE
    SYNTAX OCTET STRING
    ACCESS read-only
    STATUS mandatory
    ::= { system 1 }
```

```
OBJECT-TYPE MACRO :=
BEGIN
    TYPE NOTATION := "SYNTAX" type (TYPE ObjectSyntax)
                        "ACCESS" Access
                        "STATUS" Status
    VALUE NOTATION := value (VALUE ObjectName)
                        Access := "read-only" | "read-write" | "write-only" | "not-accessible"
                        Status := "mandatory" | "optional" | "obsolete"
END
ObjectName := OBJECT IDENTIFIER
```

Figura 7-12 Macro ASN-1 Objet-Type

Portanto, desta definição tem-se que uma classe *sOT* deve ter os atributos SYNTAX definindo a sintaxe associada, ACCESS definido o modo de acesso permitido e STATUS indicando a obrigatoriedade da implementação. Além destes atributos, cada instância de *sOT* tem um identificador de objeto correspondendo ao objeto gerenciável da MIB.

- Controle de Acesso

Cada instância de *sOT* tem os seguintes acessos permitidos:

- **ReadOnly** : Somente Leitura - disponível somente para a operações *get*.
- **WriteOnly** : Somente Escrita - disponível somente para operações *set*.
- **ReadWrite** : Escrita e leitura - disponível para operações *get* e *set*.
- **NONE** : Nenhum acesso - não está disponível para operações *get* ou *set*.

- Sintaxe

A sintaxe de um objeto gerenciável é dada por um classe derivada da classe abstrata *sOS*.

- Status

Não foi escolhida nenhuma forma de representação do atributo STATUS da macro ASN.1 *Object-Type*. Isto deve-se ao fato que este atributo tem utilidade somente de documentação. Uma vez cria a instância, não tem nenhum sentido informar se a implementação do tipo é obrigatória ou não, ou ainda, se é obsoleta.

Na figura 7.5, o diagrama de estados mostra que no estado 10, o subagente chama o método *allowGet*, da classe *sOT*, para verificar se tem permissão de acesso. Na figura 7.1, verifica-se que o controle de acesso é modelado como uma associação denominado de *authorized-on*, com um atributo de ligação *authorized*.

A figura 7.11 ilustra a declaração e a criação de dois tipos de objetos, *uPID* e *uStatus*. Verifique que os parâmetros para o construtor de *sOT* são um *string* dando sua localização completa na MIB, um apelido para o objeto e o modo de acesso.

O primeiro parâmetro, que é um *string*, informa o identificador de objeto correspondente ao tipo. O segundo parâmetro representa um apelido para o objeto. Este parâmetro ainda não tem uma utilidade na SNMPLib, mas recomenda-se que o usuário use o mesmo apelido que o tipo de objeto tem na MIB. Finalmente, o terceiro e último parâmetro identifica o modo de acesso permitido pelo tipo.

7.5 *sOS*

A classe abstrata *sOS* implementa a sintaxe de um tipo de objeto definido com a macro ASN.1 *ObjectSyntax*, mostrada na figura 7.12.

A classe *sOS*, é de uso interno da SNMPLib, apesar de estar disponível para uso. Isto é feito porque esta classe permite que a SNMPLib seja estendida com novos tipos que por ventura sejam definidos por uma nova versão da SMI. Por enquanto, o usuário somente vai usar as classes derivadas fornecidas pela biblioteca.

A SMI define várias sintaxes para tipos simples e tipos de aplicação, os quais devem ser fornecidos por uma plataforma de gerência Internet. No âmbito da SNMPLib, estas sintaxes específicas são fornecidas por classes derivadas da classes *OS*.

O quadro 7.1 resume a sintaxe da SMI e as classes que as implementam e são fornecidas pela SNMPlib.

Tipo Definido pela SMI	Classe
INTEGER	sOSInteger
OCTET STRING	sOSString
OBJECT IDENTIFIER	sOID
IpAddress	sOSIpAddress
NetWorkAddress	sOSNetWorkAddress
Counter	sOSCounter
Gauge	sOSGauge
TimeTicks	sOSTimeTicks
DisplayString	sOSDisplayString
NULL	N/A
Opaque	N/A

Quadro 7-1 Classes que Implementam Tipos Pré-Definidos na SMI

7.6 sOID

A classe *sOID* implementa um identificador de objeto como definido pela SMI da Internet [20]. Um identificador de objeto é uma seqüência de números inteiros não-negativos, por exemplo

1.0.234.10.999.4

Na SNMPlib, a classe *sOID* é uma implementação do tipo abstrato de dados Seqüência. Uma instância de Seqüência mantém uma relação de ordem, permitindo que se estabeleça uma relação de anterior ou próximo entre seus elementos. Além disso, cada elemento tem uma posição única dada por um número inteiro n , onde $0 < n \leq lenght()$.

Na descrição a seguir, considere que *o1* e *o2* são instâncias da classe *sOID*, *n*, *p* e *i* são números inteiros quaisquer.

- ***elementAt*** : a chamada

$$n = o1.elementAt(p)$$

retorna em *n*, o elemento na posição *p* de *o1*.

- ***addAtPosition*** : a chamada

$$o1.elementAtPosition(i,p)$$

coloca o elemento *i* na posição *p* da instância.

- ***head***: retorna o primeiro elemento da instância, ou seja,

$$o1.head() = o1.elementAt(1)$$

- ***tail*** : retorna o último elemento da instância, ou seja,

$$o1.tail() = o1.elementAt(o1.length())$$

- ***shift***: remove o primeiro elemento da instância e desloca os elementos restantes, no sentido da primeira posição, de tantas posições quanto forem dadas pelo parâmetro de entrada de *shift*.

Por exemplo, no identificador 1.0.234.10.999.4, a aplicação de *shift(2)* teria como resultado 234.10.999.4.

- ***length***: retorna quantos elementos possui a instância.
- ***equal***: verifica a igualdade entre duas instâncias de *sOID*.

Se

$$o1.equal(o2) == Verdade, \text{ então } o1.length() = o2.length$$

e

para *i* de 0 até *o1.length()*,

$$o1.elementAt(i) = o2.elementAt(i)$$

- **sub**: a chamada

$$o2 = o1.sub(n)$$

retorna em $o2$, os n primeiros elementos de $o1$, sendo que $o2$ contém somente n elementos, ou seja, $o2.length() = n$.

- **diff**: efetua a operação de diferença, ou seja:

$$\text{Considere que } o3 = o1.diff(o2)$$

Então esta operação tem como pré-condições:

1. $o1.length() \geq o2.length()$
2. $o1.sub(o2.length()) = o2$

O resultado será dado por

$$o3 = o1.shift(o2.length())$$

Portanto, sendo

$$o1 = "2.6.2.4.1.2.1.5.9"$$

$$o2 = "2.6.2"$$

$$o3 = "1"$$

$$o4 = ""$$

resulta que

$$o1.diff(o2) = "4.1.2.1.5.9".$$

$$o3.diff(o4) = o3$$

7.7 *sModuleMIB*

A classe *sModuleMIB* modela uma base de dados contendo referências aos tipos de objetos controlados pelo subagente. O subagente pode utilizar os métodos disponíveis na interface pública de *sModuleMIB* para obter informações sobre os tipos de objetos e sobre os objetos sintaxes associados. Isto acontece, por exemplo, no método *dispatch* de *sAgent* que antes de chamar os métodos *procGet* ou *procSet*, faz uma verificação se estão corretas e válidas, as informações sobre o tipo de objeto e a operação contidas na SNMP PDU. Por enquanto, *sModuleMIB* tem somente dois atributos, a árvore privada e um identificador de objeto apontando para a raiz da MIB privada.

A árvore privada é a implementação de um tipo abstrato de dados que mantém uma relação de ordem entre seus elementos. Desta forma, existe uma relação de anterior ou próximo entre as instâncias da classe *sOT*.

A interface pública da classe *sModuleMIB* define basicamente três métodos de acesso ao atributo árvore privada:

- ***addOT*** - Permite que o cliente da classe adicione, após a posição corrente, um novo tipo de objeto na árvore privada.
- ***locateOT*** - Localiza um tipo de objeto, sendo fornecido o identificador. Depois desta chamada, o atributo posição corrente aponta para a instância localizada.
- ***removeOT*** - Permite que o cliente da classe remova o tipo de objeto apontado pelo atributo posição corrente.

Além destes métodos, foram definidos outros métodos: *load* que carrega, de uma única vez, todo o módulo MIB a partir de listas de objetos fornecidas como parâmetros, e *halt*, que remove todo o módulo.

7.8 *sSMUX*

sSMUX é a classe que proporciona um meio de comunicação entre o subagente e o agente SNMP utilizando o protocolo SMUX.

Através dos métodos públicos disponíveis em *SMUX*, o subagente pode:

1. Estabelecer uma conexão *SMUX* com o agente *SNMP*.
2. Pedir o registro de uma subárvore privada.
3. Receber PDUs.
4. Enviar PDU *Get-Response*.
5. Enviar *traps*.
6. Encerrar a conexão *SMUX* com o agente *SNMP*.

• Estabelecimento de uma Conexão *SMUX* com o Agente

O método *open* abre uma conexão entre o subagente e o agente *SNMP*. A abertura de uma conexão é simples: o método *open* envia uma PDU *SMUX OpenPDU* para o agente. Se o agente recusar a conexão, ele insere uma PDU *SMUX ClosePDU* e fecha a conexão. Caso contrário, nenhuma resposta é enviada.

O arquivo “*/etc/snmpd.conf*” define parâmetros de configurações para o agente *SNMP* e para os subgentes *SMUX* implementados no sistema. Entre estas configurações, estão entradas para cada subagente, definindo o seu identificador de objeto e uma senha usada para autenticação.

Desta forma, o método *open* necessita de três parâmetros:

- **Identidade** : é o Identificador de Objeto deste subagente.
- **Descrição**: é um *string* descrevendo o subagente.
- **Senha**: É a senha do subagente.

O método *open* somente envia a *SMUX PDU OpenPDU* e imediatamente retorna da chamada. Se uma *SMUX PDU ClosePDU* for enviada, ela é recebida e tratada no subagente pelo método *dispatch*.

• Pedido do Registro Subárvore Privada

O método *registerTree* permite que o subagente envie, para cada subárvore por ele controlada, uma PDU *SMUX RReqPDU* para registrá-la. Para cada *RReqPDU* recebida e aceita pelo agente, ele envia uma *PDU RRspPDU* para o subagente.

O método *registerTree* somente envia uma PDU *RReqPDU* e imediatamente retorna da chamada. Cada subsequente *RRspPDU* recebida como resposta, é tratada no subagente pelo método *dispatch*.

É importante notar que cada pedido de registro de uma subárvore é acompanhado por um valor que varia entre 0 e $2^{31} - 1$ definindo a prioridade de registro. Quanto menor o valor, maior é a prioridade. Múltiplos agentes podem registrar a mesma subárvore, mas somente aquele com maior prioridade é exclusivamente consultado para todas as operações naquela subárvore.

Outro parâmetro informado é a operação desejada. A operação desejada pode ser um pedido para deleção do registro da subárvore no agente, um pedido para registro de uma subárvore onde o subagente tem permissão de leitura e, finalmente, um pedido de registro onde o subagente tem permissão de leitura e escrita sobre os objetos da MIB.

Desta forma, o método *registerTree* necessita de três parâmetros:

- Subárvore : é o identificador de objeto da raiz da subárvore controlada pelo subagente.
- Prioridade : é a prioridade do pedido.
- Operação : é a operação desejada, que pode ser *del* para pedir uma operação de deleção de registro de subárvore, *RO* para pedir um registro com modo de acesso de somente leitura e *RW* para pedir um registro com modo de acesso de leitura e escrita na subárvore.

Deve ser notado que um efeito denominado de **montagem** pode ocorrer. Este efeito “esconde” os tipos registrados, pela instância de objeto em questão, para outras instâncias de subagentes. Por exemplo, se um subagente registra a subárvore *sysDescr* e mais tarde, outro subagente registra a subárvore *system* (cujo um dos ramos é a subárvore *sysDescr*), então todos os pedidos para *sysDescr* serão enviados para o último subagente.

• Receber PDUs

O método *wait* permite que o subagente fique esperando a chegada de uma PDU na fila de mensagens da interface SMUX. A fila de mensagens é uma estrutura de dados onde o subagente armazena as PDUs enviadas pelo agente. Estas PDUs dizem respeito à tipos de objetos registrados pelo subagente SMUX.

O método *wait*, depois de chamado, verifica se há alguma PDU na fila. Se houver, então *wait* copia o conteúdo da primeira PDU da fila, para a instância enviada como parâmetro. Em seguida, *wait* remove a PDU da fila e retorna a chamada. Se, do contrário, não há nenhuma PDU no momento da chamada, então o método fica em estado de suspensão, aguardando a chegada de uma.

- **Enviar PDU *Get-response***

O método *response* permite que o subagente envie uma PDU contendo uma mensagem *Getobservação -response*. A mensagem *Get-response* contém os valores associados aos tipos de objetos pedidos em uma operação *Get* anterior.

- **Enviar *Traps***

O método *trap* permite que o subagente envie um trap, informando alguma condição anormal detectada em algum objeto gerenciável.

A chamada para o método *trap* deve ser feita fornecendo-se como parâmetro uma PDU de *trap*. A PDU de *trap*, poderá incluir uma lista de tipos de objetos como forma de identificar sobre quais atributos do objeto gerenciável, a condição anormal é verificada.

- **Encerrar a conexão SMUX com o agente**

O método *close* permite ao subagente encerrar a conexão SMUX com o agente. Este encerramento é imparcial, ou seja, o subagente pode, em qualquer momento, encerrar a conexão, independente do estado da conexão SMUX.

O método *close*, na sua chamada, necessita de um número inteiro como parâmetro. Este número informa a razão do encerramento e pode ter, tanto para o agente SNMP, quanto para o subagente, os seguintes valores:

- ***goingDown*** : Informa tanto para o agente, quanto para o subagente o seu encerramento.
- ***unsupportedVersion*** : Informa uma versão de protocolo não suportada. O agente envia uma *ClosePDU*, indicando a versão com erro.

- **packetFormat** : A PDU recebida foi codificada incorretamente ou foi corrompida na transmissão, resultando em um erro de codificação.
- **protocolError** : Uma seqüência incorreta de PDUs ou uma PDU não esperada foi recebida.
- **internalError** : Um erro fatal ocorreu no processamento de uma PDU.
- **authenticationFailure** : O agente envia uma *ClosePDU*, indicando que o identificador de objeto e a senha contidos na *OpenPDU*, não foram encontrados no arquivo de configuração local.

Tanto o agente, quanto o subagente podem iniciar o término da conexão. Mas somente o agente pode informar *unsupportedVersion* e *authenticationFailure* como razão de encerramento.

A figura 7.13 mostra o diagrama de estados da classe *SMUX*. Verifique que no estado *IDLE*, não existe um método sendo chamado. Uma instância da classe neste estado pode receber a chamada de um método *trap* ou *wait*. Entretanto, pode ocorrer a chegada de uma PDU. Neste caso, a PDU é colocada na fila para ser retirada pela chamada de um método *wait*.

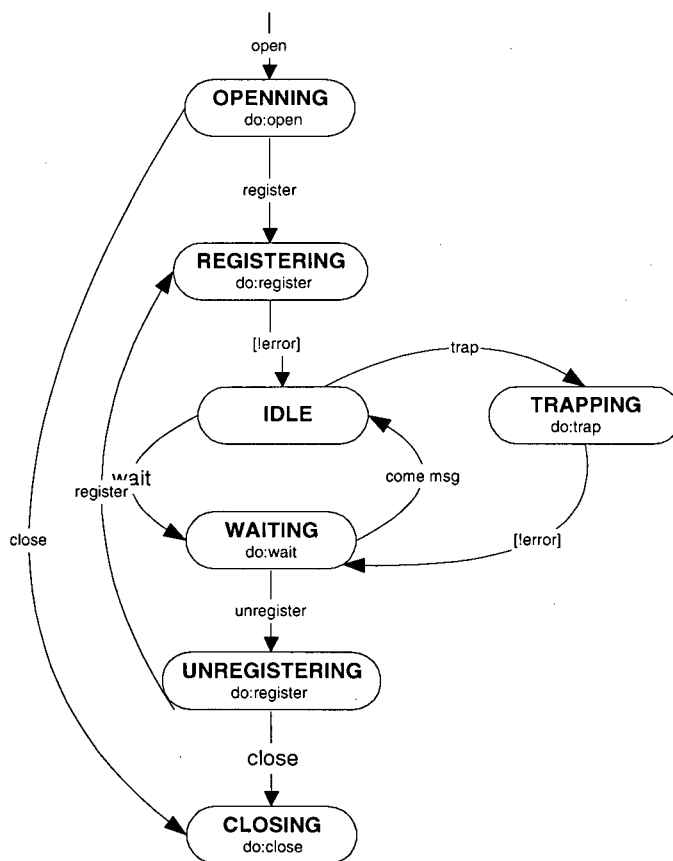


Figura 7-13 Diagrama de Estado da Classe *sSMUX*

7.9 *sTE*

Cada instância da classe *sTE* representa um tipo de dado ASN.1, em uma forma independente de máquina. Esta estrutura interna é intercambiável entre aplicações que estão sendo executadas em nós da rede, heterogêneos ou não.

7.10 *sVarBind*

A classe *sVarBind* implementa uma estrutura contendo uma instância da classe *sOID* e uma instância da classe *sTE*.

Como pode ser observado no diagrama de classes da Figura 7.1, a classe *PDU* contém uma relação de agregação com a classe *sVarBind* de um para muitos. De fato, cada instância de *sPDU* contém uma lista de instâncias *sVarBind* que servirão como parâmetros para a operação SNMP.

7.11 *sPDU*

A classe *sPDU* implementa uma estrutura de dados que encapsula as informações contidas em uma operação ou *trap* SNMP, definindo métodos para manipular esta estrutura.

Os atributos da classe *sPDU* são definidos como:

- *request-id*

request-id é do tipo inteiro. É usado pelo gerente para distinguir os pedidos feitos, permitindo que o gerente possa enviar um próximo pedido sem necessitar obter antes, a resposta do pedido anterior. As respostas que chegam podem, então, ser relacionadas com os pedidos correspondentes feitos. Outra aplicação é evitar que possam haver mensagens e operações duplicadas na rede.

- *error-status*

Se o valor deste campo é diferente de zero, ele indica uma exceção ocorrida no processamento do pedido. Os possíveis valores são:

1. **tooBig**: o subagente não pôde preencher uma operação dentro da mensagem SNMP.
2. **noSuchName**: a operação pedida identificou um nome de variável desconhecida, levando em consideração o perfil (*profile*) da comunidade.
3. **badValue**: a operação requisitada especificou uma sintaxe ou um valor incorreto quando tentando modificar uma variável.
4. **readOnly**: a operação requisitada tentou modificar uma variável que, de acordo com o perfil da comunidade, não tem permissão de escrita.
5. **genErr**: qualquer outro erro, que não sejam os acima listados.

- **error-index**

É do tipo inteiro. Se o seu valor for diferente de zero, indica qual variável no pedido tem um erro. Este campo é diferente de zero unicamente para erros *noSuchName*, *badValue*, *readOnly*. Neste caso, é o deslocamento dentro do campo *variable-bindings* (a primeira variável é dita ser de deslocamento 1).

- **variable-bindings**

É uma lista de variáveis contendo instâncias da classe *VarBind*, ou seja, entre a classe *sVarBind* e *sPDU* existe uma relação de agregação como pode ser notada na figura 7.1.

Pode ser verificado, na definição da classe *sVarBind*, que uma instância desta contém um identificador de objeto e um elemento de transmissão. O elemento de transmissão associado a cada instância *sVarBind* não é significativa para PDUs *Get-request* e *Get-Next-request*. O subagente deverá simplesmente ignorar o elemento de transmissão fornecido pela entidade enviando o pedido SNMP.

- ***type-message***

Indica o tipo de mensagem SMUX que a PDU contém. Os tipos possíveis de mensagens podem ser:

- ***Register-response***: contém o resultado de um pedido anterior para registrar uma subárvore, feito através do método *registerTree*.
- ***Get-request*** : pede ao subagente para que seja efetuado a operação *Get* tomando como parâmetro *variable-bindings*.
- ***Get-Next-request*** : pede ao subagente para que seja efetuado a operação *Get-next* tomando como parâmetro *variable-bindings*..
- ***Set-request*** : pede ao subagente para que seja efetuado a operação *Set*, tomando como parâmetro *variable-bindings*. Entretanto, a operação *Set* somente é preparada, colocando os parâmetros da operação em uma lista de operações *Set* pendentes. Numa fase posterior, o agente, através de uma mensagem *CommitOrRollBack*, pode confirmar se realmente pode ser realizada ou não esta operação pelo subagente.
- ***CommitOrRollback-request***: o agente indica ao subagente que a operação *Set* iniciada com uma operação *set* anterior, deve ser deferida ou não.
- ***Close-request***: o agente informa ao subagente que encerrou, imparcialmente, a conexão entre os dois. O subagente deve também encerrar a conexão de sua parte.

7.12 *sTrap*

A classe *sTrap* define uma estrutura para conter as informações de uma PDU *Trap* e define métodos para manipular esta estrutura.

Os atributos de uma instância de *sTrap* são:

- ***enterprise***

É um identificador de objetos que contém o valor do *sysObjectID* do agente.

- ***agent-addr***

agent-addr é o valor do *NetworkAddress* do agente.

- ***generic-trap***

É um número inteiro que pode ter os seguintes valores e significados:

1. ***coldStart***: o agente é reinicializado e os objetos sobre seu controle poderão ser alterados.
2. ***warmStart***: o agente é reinicializado e os objetos sobre seu controle não poderão ser alterados.
3. ***linkDown***: uma das interfaces de rede mudou o estado para DOWN, identificada pela primeira variável *notrap*.
4. ***linkUp***: uma das interfaces de rede mudou o estado para UP, identificada pela primeira variável no *trap*.
5. ***authenticationFailure***: ocorreu uma falha de autenticação numa entidade SNMP.
6. ***egpNeighborLoss***: um par EGP mudou para o estado DOWN, identificado pela primeira variável *notrap*.
7. ***enterpriseSpecific***: informa outros tipos de *traps*, identificados no campo *specific-trap*.

- ***specific-trap***

O valor de *specific-trap* é um número inteiro. Se o valor é diferente de zero, identifica qual o *trap enterpriseSpecific* que ocorreu.

- ***time-stamp***

É o valor do objeto *sysUpTime* do agente quando o *trap* ocorreu.

- ***variable-bindings***

variable-bindings indica uma lista de variáveis contendo informação sobre *otrap*.

7.13 *sPending*

Cada instância desta classe é um par de instâncias de *sOT* e *sOS*.

Como pode ser notado na figura 7.1, existe uma associação de “um para muitos” entre as classes *sOT* e *sOS* denominada *setPendings*. Cada link dessa associação é uma instância da classe *sPending*. Cada uma dessas instâncias indica uma operação *Set* pendente esperando um confirmação através de uma mensagem *CommitOrRollback* para realmente ser realizada ou rejeitada.

8. BIBLIOTECA DE CLASSES DE REDES NEURAIIS - NEUROLIB

Devido às características das redes neurais, o uso de OMT para a construção de um conjunto de classes que implementem redes neurais é muito apropriado. Através da OMT, consegue-se capturar aspectos relevantes das classes e aplicá-los numa implementação com uma linguagem orientada à objetos. Estes aspectos, como serão vistos adiante, usam muito herança, uma das fortes características da **Biblioteca de Classes de Redes Neurais - Neurolib**.

A construção da biblioteca foi baseada nas classes desenvolvidas por Adam Blum [04]. Basicamente, as classes aqui apresentadas foram modificações e extensões feitas a estas classes, para se adequarem à análise e projeto feitos pela metodologia OMT e, além disso, suprirem algumas necessidades de generalização necessárias no domínio de gerência de redes de computadores.

8.1 Hierarquia de Classes

A hierarquia de classes da Neurolib é relativamente simples. Ela se baseia em três grandes classificações dos modelos de redes neurais [08][12][26][33]:

- **Modelos com Treinamento Supervisionado** - No treinamento supervisionado, para cada vetor de entrada apresentado à rede, há um vetor representando a saída desejada, ambos formando um par de treinamento. Exemplo de tais redes são as redes com algoritmo de treinamento *Backpropagation* [25].
- **Modelos com Treinamento Não Supervisionado** - O treinamento não-supervisionado não necessita de nenhum vetor de saída desejada e portanto, nenhuma comparação é

realizada entre a saída obtida e uma saída desejada. O conjunto de treinamento consiste inteiramente no vetores de entradas. Exemplos de tais redes são as Redes Neurais de *Kohonen* e *Grossberg* [09].

- **Modelos Híbridos** - são modelos formados à partir de agregação de redes neurais com modelos que podem ter treinamento supervisionado e não supervisionado. Exemplos de tais redes podem ser as *Counterpropagation* [04][08].

Portanto, podemos formar o diagrama de classes da figura 8.1, representando as mais importantes características das classes. Neste diagrama, as classes com os modelos (templates) tracejadas indicam que sua implementação não está disponível na Neurolib.

8.2 *nNet*

nNet é a classe que reúne as características comuns de todas as outras classes de redes neurais. A classe *nNet* é uma classe abstrata, onde todas as outras classes de redes neurais são derivadas.

- *tolerance*

É um valor de ponto flutuante, representando a fração (valor entre 0 e 1) dentro do qual a rede neural aceita uma saída como válida.

- *ninputs*

É o número de neurônios da camada de entrada, no caso da Neurolib, representa o número de elementos do vetor de entrada.

- *noutputs*

É o número de neurônios da camada de saída, no caso da Neurolib, representa o número de elementos do vetor de saída.

- *learnrate*

É a taxa de aprendizagem da matriz de pesos (ou sinapses) .

- *recall*

O método abstrato *recall* executa a rede, ou seja, o método recebe como parâmetro um vetor de entrada e calcula o vetor de saída.

O método *recall* é abstrato, tendo em vista que sua implementação é dependente do modelo adotado. Por exemplo, com uma rede BAM, o resultado é dependente da matriz de correlação encontrada, enquanto nas redes *Backpropagation*, o resultado depende da matriz de pesos formada no treinamento.

- *activation*

O método *activation* modela a função de ativação de um rede neural.

A função de ativação é uma função $f: x \rightarrow y$. Desta forma, o método *activation* deve retornar o valor da aplicação de um função de ativação no parâmetro de entrada.

A classe *nNet* fornece como função de ativação implícita, a função sigmóide:

$$f(x) = \frac{1}{1 + e^{-x}}$$

A figura 8.2 mostra um exemplo da implementação, em C++, de uma função de ativação que usa a tangente hiperbólica. Nota-se que o método *activation* foi implementado em uma classe derivada da classe *Backpropagation*, que não é uma classe abstrata.

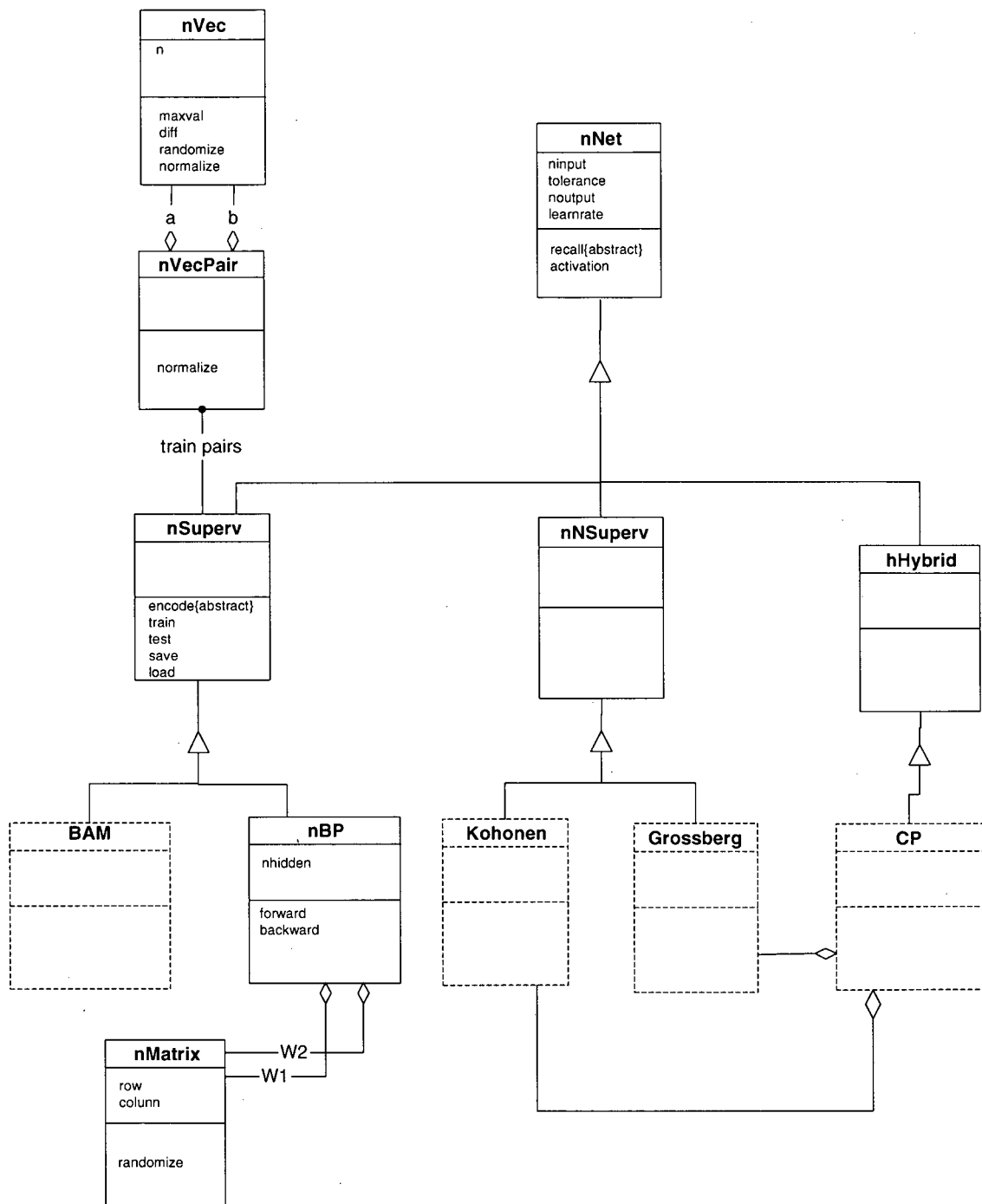


Figura 8-1 Diagrama de Classes da Biblioteca de Redes Neurais

```
    .  
    .  
extern float tanh(float x);  
float Backpropagation::activation(float x)  
{  
    return tanh(x)  
}  
    .  
    .
```

Figura 8-2 Implementação do Método *activation*

8.3 *nNSuperv*

A classe *nNSuperv* é uma classe abstrata e representa todos os modelos de redes com treinamento não supervisionado. Nestas redes, os ajustes das sinapses não são baseados em comparações com alguma saída desejada, ou seja, a rede é auto-organizada.

Esta classe aparece na modelagem por razões de completude. Por enquanto é somente é um alias para a classe *nNet*.

8.4 *nSuperv*

A classe *nSuperv* representa os modelos de redes neurais com treinamento supervisionado. Nestas redes, os resultados obtidos da aplicação de uma entrada na rede neural, são comparados com uma saída desejada. Os erros assim obtidos são usados para ajustar os pesos (sinapses) da rede.

O diagrama de estados desta classe é representado pela figura 8.3 e o diagrama de fluxo de dados está representado na figura 8.4. No diagrama da figura 8.4, *net* representa uma instância da classe *nSuperv*.

Pode ser observado que a rede possui dois estados finais: SUSPEND e TRAINNED. O estado SUSPEND representa a rede que no seu processo de treinamento (processo *train* na figura 8.4), atingiu o número máximo de iterações sem conseguir apreender. O estado TRAINNED representa a rede que conseguiu ser treinada antes de atingir o número máximo de iterações.

Além disso, observa-se na figura 8.4, que o processo *train* retira pares de vetores do conjunto de treinamento, representado pelo depósito de dados *train-pairs*.

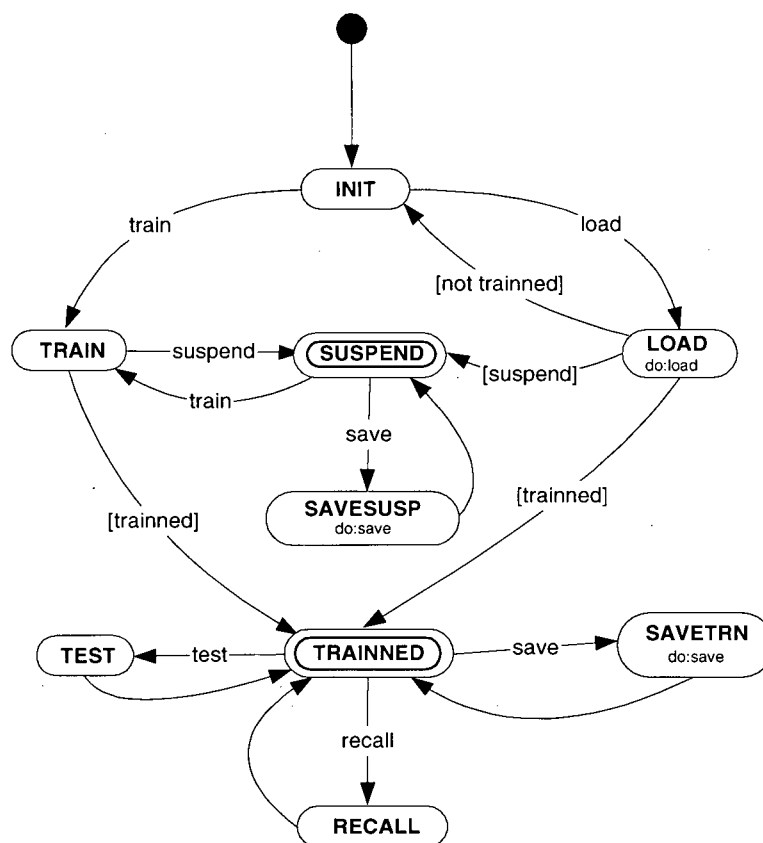


Figura 8-3 Diagrama de Estados para a Classe Superv

O estado RECALL, na figura 8.3, é atingido quando o processo *recall*, na figura 8.4, está ativo. Este processo executa a rede, ou seja, recebe como parâmetro um vetor de entrada (*in*) e produz um vetor de saída (*out*).

LOAD é o estado que se encontra uma instância de *nSuperv* quando executa o método *load* e, os estados SAVESUSP e SAVETRAN quando executam *save*. Estes dois métodos são implementações dos processos *load* e *save* do diagrama de fluxo de dados da figura 8.4. O processo *load* lê as definições (entrada *parms*) de uma instância da classe *nSuperv* e *save*, grava as definições em um arquivo.

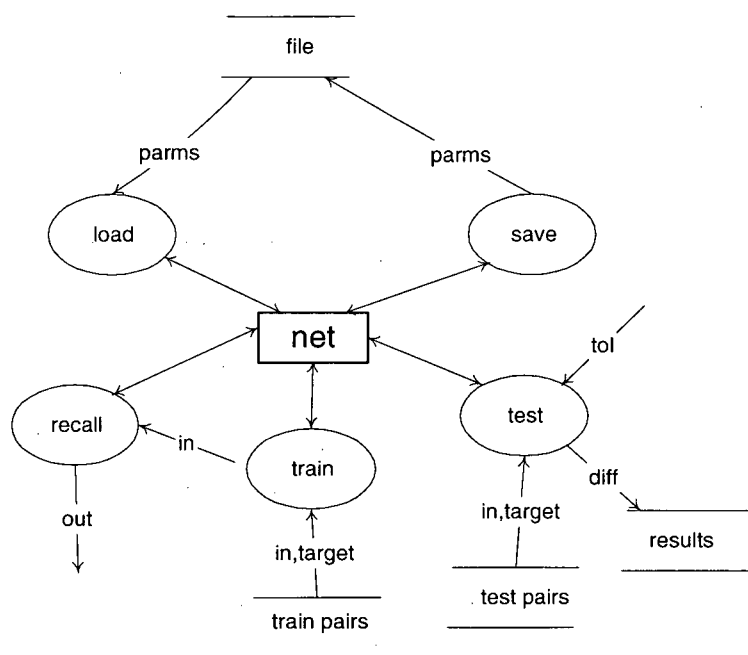


Figura 8-4 Diagrama de Fluxo de Dados para a Classe nSuperv

O estado TEST é o estado alcançado quando o processo *test* está ativo. O processo *test* serve para testar a rede uma vez treinada. Desta forma, pares de vetores, cada um com um vetor de entrada e outro de saída desejada, são injetados na rede. Ao término do processo, se tem a diferença obtida entre o vetor de saída desejado e o vetor de saída obtida na rede, para cada um dos pares de teste. Com isto, estas diferenças podem ser comparadas com a tolerância, verificando se a rede acertou a saída ou não.

- Método *train*

O método abstrato *train* implementa o treinamento da rede neural. Ele é abstrato, portanto deve ser definido pelas classes derivadas da classe *nSuperv*.

A figura 8.5 mostra o diagrama de fluxo de dados do processo *train* que é a implementação do método *train*. A figura 8.5 também mostra o processo *cycle*, contido no processo *train*.

Os parâmetros do método *train* são o conjunto de treinamento e o número máximo de iterações. O conjunto de treinamento é uma lista de pares de vetores. Cada par contém o vetor de entrada (*in*) e o vetor de saída desejado (*target*).

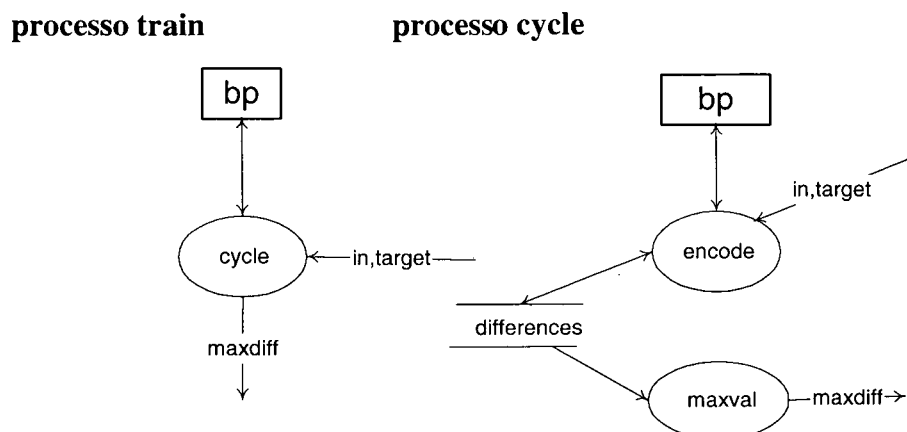


Figura 8-5 Diagrama de Fluxo de Dados do Processo *train*

Na figura 8.6 está representado o diagrama de estados do estado TRAIN. O número máximo de iterações (*maxiters*) é o número limite de iterações que o processo de treinamento pode realizar. Se a rede não conseguir treinar com a tolerância desejada (*tol*), o processo de treinamento é interrompido (evento *suspend*) e o método retorna, ficando a rede com a matriz de pesos da última iteração.

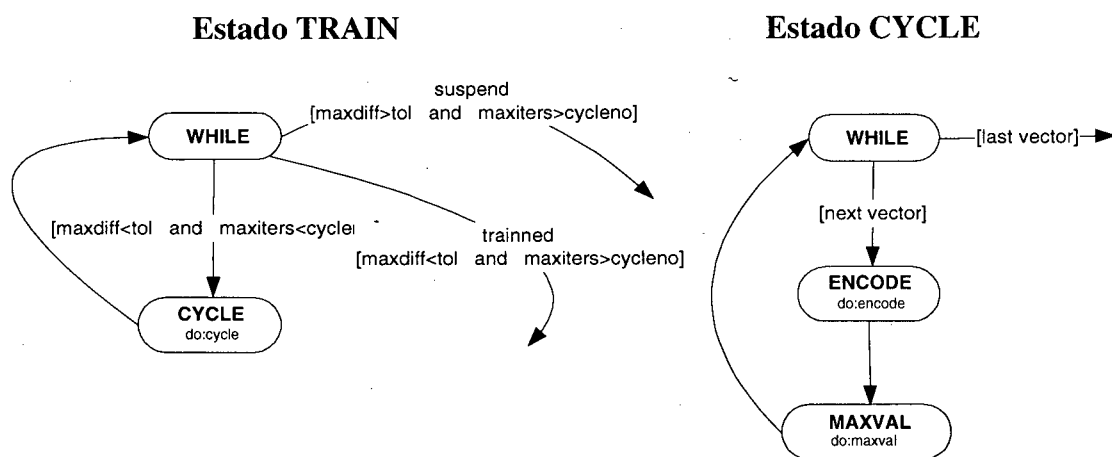


Figura 8-6 Diagrama de Estados do Estado TRAIN

- Método *encode*

O método *encode* é o método que:

- Aplica a função de ativação à cada vetor de entrada do par de treinamento, produzindo um vetor de saída.

- Com o vetor de saída obtido, calcula o erro e ajusta a matriz de pesos.

O método *encode* é definido como abstrato e deve ser fornecido pelas classes derivadas de *nSuperv*. Os parâmetros de entrada para o método são o vetor de entrada e o vetor de saída desejada.

- Método *load*

O método *load* carrega uma rede, salva pelo método *save*, de um arquivo e inicializa a rede neural com os valores lidos.

A partir do momento que a operação *load* terminar, a rede está exatamente no estado que ela estava quando foi salva (figura 8.3).

- Método *save*

Este método tem por função salvar em um arquivo os parâmetros de definições e a matriz de pesos da rede neural. Desta forma, a rede pode ser carregada em um outro momento e ser executada, ou ainda, ser transportada para ser executada em outra máquina (figuras 8.3 e 8.4).

8.5 *nBP*

A classe *nBP* é derivada da classe *nSuperv* e implementa o modelo de redes neurais com treinamento supervisionado usando o algoritmo *Backpropagation*.

O conceito envolvido no *Backpropagation* é muito genérico e existem muitas variantes do algoritmo e de implementações. Entretanto, a implementação adotada segue, com algumas alterações, a proposta de Adam Blum [04]. Esta implementação tem a topologia do modelo apresentado na figura 8.7.

Como pode ser visto, este modelo tem uma topologia em três camadas: camada de entrada, camada oculta e camada de saída. Há duas matrizes de pesos ou sinapses: a matriz $W1$,

entre a camada de entrada \mathbf{i} e a camada oculta \mathbf{h} , e a matriz W_2 , entre a camada oculta \mathbf{h} e a camada de saída \mathbf{o} .

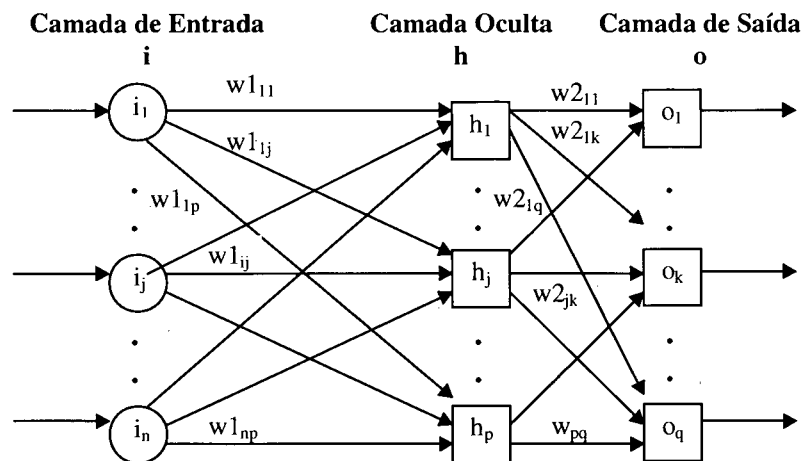


Figura 8-7 Topologia do *Backpropagation*

- Atributo *momentum*

O atributo *momentum* é um valor, entre 0 e 1, indicando qual a quantidade da variação do peso anterior é tomada para mudar o peso corrente.

- Atributos i, h e o

Os atributos i, h e o são atributos da classe $nVec$, representando os neurônios das camadas de entrada, oculta e saída.

- Atributos $h1$ e $h2$

$h1$ e $h2$ são os vetores *threshold* para os neurônios da camada oculta \mathbf{h} e para os neurônios da camada de saída \mathbf{o} , respectivamente. No momento da construção de uma instância da classe nBP , aos elementos destes vetores *threshold* $h1$ e $h2$ são atribuídos valores randômicos entre -1 e +1.

- Atributos $W1$ e $W2$

$W1$ e $W2$ são as matrizes de pesos entre as camadas i e h e entre as camadas h e o , respectivamente. No momento da construção de uma instância da classe `Backpropagation`, aos elementos das matrizes de pesos $W1$ e $W2$ são atribuídos valores randômicos entre -1 e $+1$.

- Método `encode`

O método `encode` implementa os passos *forward* e *backward* do Algoritmo `Backpropagation` [42], como pode ser descrito pelos Diagramas de Estados da Figura 8.8 e de Fluxo de Dados da figura 8.9. Como pode ser observado, o método `encode`, que é a implementação do processo `encode`, compõe-se de dois processos: o processo *forward* e o processo *backward*. A figura 8.10 mostra o diagrama de estados para os estados FORWARD e BACKWARD.

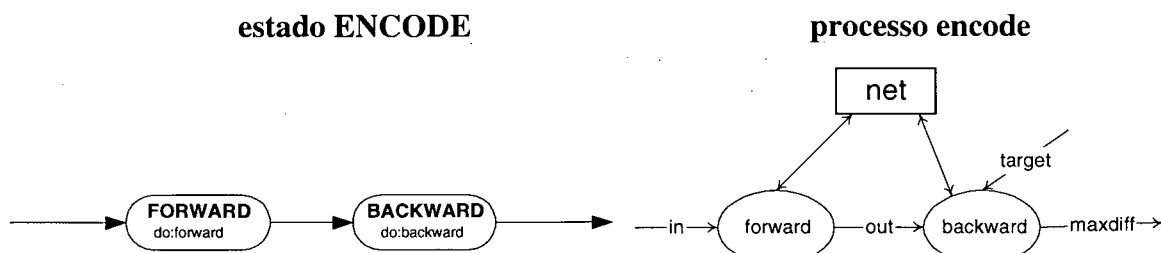


Figura 8-8 Diagrama de Estados

Figura 8-9 Diagrama de Fluxo de Dados

O processo *forward* é a implementação do passo *forward* do algoritmo `Backpropagation`. O Diagrama de Fluxo de Dados da figura 8.11 apresenta a modelagem do processo *forward*.

Como foi exposto no capítulo 3, o passo *forward* consiste no cálculo de $F(X.W)$ para as camadas oculta e de saída:

1. Para a camada oculta: $h = F(\text{in}.W1 + \text{threshold})$
2. Para a camada de saída: $o = F(h.W2 + \text{threshold})$

O processo *backward* é a implementação do passo *backward* do algoritmo `Backpropagation`, podendo ser modelado pelo diagrama do estado BACKWARD da figura 8.9 e do diagrama de fluxo de dados do processo *backward* da figura 8.12.

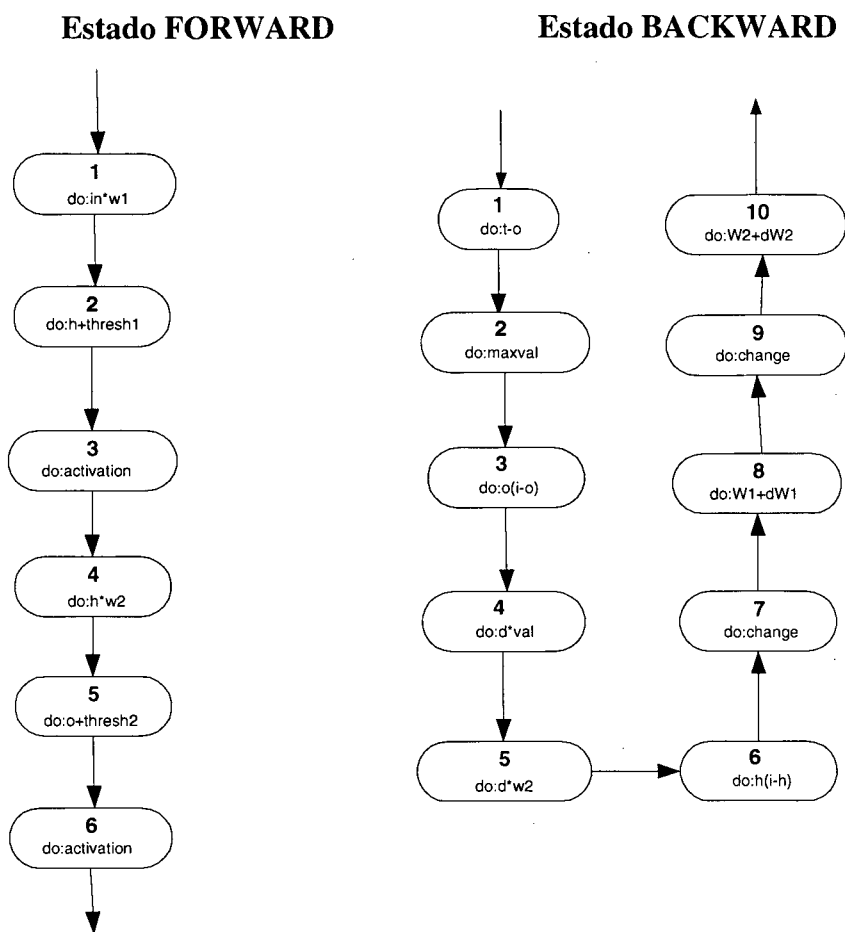


Figura 8-10 Diagrama de Estados dos Estados FORWARD E BACKWARD

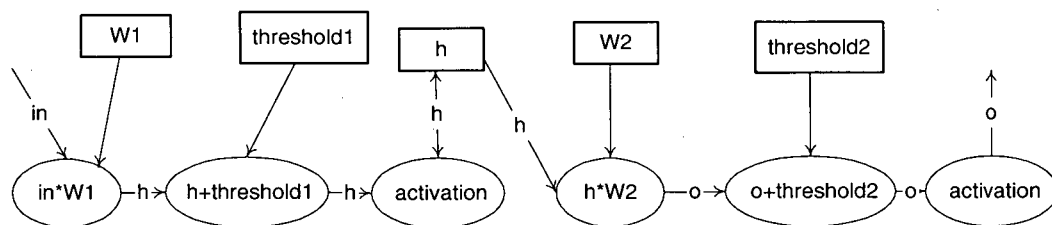


Figura 8-11 Diagrama de Fluxo de Dados do processo forward

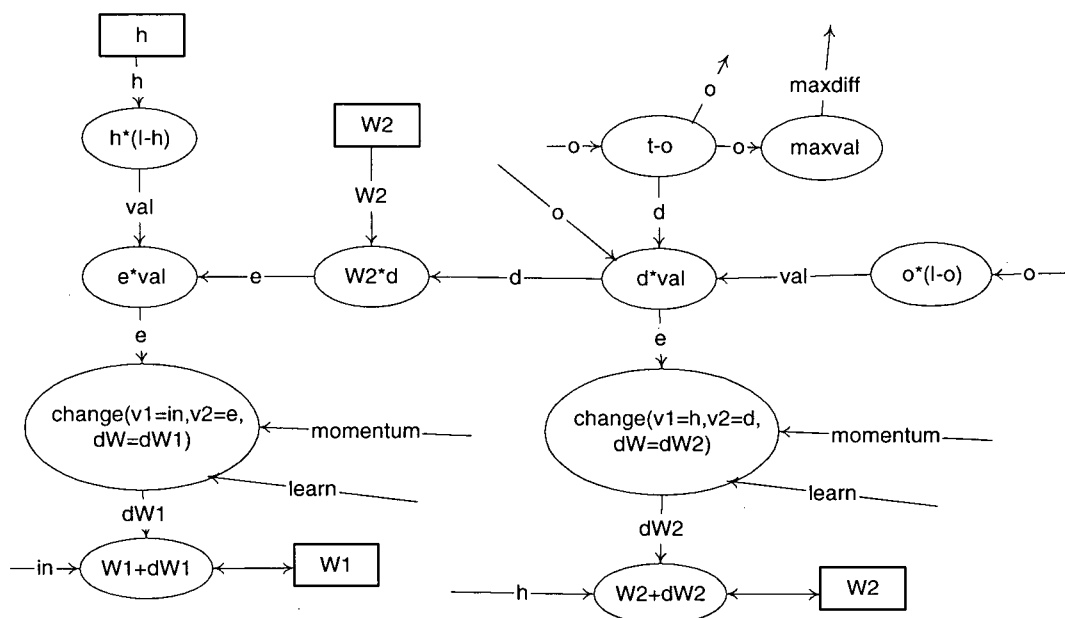


Figura 8-12 Diagrama de Fluxo de Dados do Processo Backward

O processo *backward* envolve os seguintes passos:

1. Cálculo do Erro da Camada de Saída: $d = o(I-o)(o-target)$
2. Cálculo do Erro na Camada Oculta: $e = h(I-h)W2d$
3. Ajuste dos Pesos dos Neurônios da Camada de Saída: $W2 = W2 + dW2$
4. Ajuste dos Pesos dos Neurônios da Camada Oculta: $W1 = W1 + dW1$

9. PROGRAMA PARA PROTOTIPAÇÃO E TREINAMENTO DE REDES NEURASIS - NEUROPROTO

O Programa para Prototipação e Treinamento de Redes Neurais é um programa que auxilia no projeto de redes neurais e na validação de seu uso como solução para um determinado problema. Além disso, através de uma série de testes pode-se encontrar as melhores configurações.

Este programa é implementado com as classes da Neurolib. Como consequência, o ambiente pode ser utilizado para treinamento de uma rede neural, sendo desnecessária a implementação dos procedimentos de treinamento no agente adaptivo. Uma vez treinada a rede, ocorre o salvamento das sinapses e a transferência das mesmas para o agente adaptivo. Além disso, o agente pode ser atualizado com um novo treinamento sempre que for necessário.

9.1 Descrição do NeuroProto

O Programa para Prototipação e Treinamento de Redes Neurais, NeuroProto, foi implementado, assim como as bibliotecas, em um sistema operacional UNIX, usando o ambiente gráfico OSF CDE (*Common Desktop Environment*) no Motif 1.2. Portanto, o programa pode ser executado na maioria dos sistemas operacionais mais recentes compatíveis com o UNIX System V e que executam o OSF CDE, como IBM AIX 4 e o Sun Solaris.

Assim que é chamado para executar, o programa exibe a janela da figura 9.1. Por esta figura, percebe-se que a janela principal do ambiente é caracterizado pelas seguintes áreas:

- Barra Superior

- Barra de Botões
- Área do Gráfico
- Área de *Logs*
- Área de *Status*

9.1.1 Barra Superior

Pode-se perceber, pela figura 9.1, que na parte superior existe uma barra com dois itens de menus:

- *File* - Este item é usado para carregar as definições da rede neural e salvar as sinapses da rede neural. Assim que for selecionado o item, um *submenu* é exibido, contendo dois itens:
 - *Open* - Carrega as definições de uma rede neural contidas no arquivo de definição e as sinapses, se a rede neural foi salva anteriormente.
 - *Save* - Salva as sinapses da rede neural.
- *Help* - Através deste item de menu, obtém um auxílio rápido para determinadas tarefas do sistemas, tais como definir uma rede neural, efetuar o treinamento e o significado do gráfico.

9.1.2 Botões

Na parte inferior da janela, existem vários botões:

- *Train* - Este botão faz com que a rede comece o treinamento, se ainda não estiver treinada.
- *Suspend* - Este botão faz com que a rede suspenda o processo de treinamento. As matrizes de pesos ficam com os valores da última iteração efetuada.
- *Resume* - Reassume o treinamento de uma rede neural a partir do última iteração efetuada antes do processo ser suspenso.
- *Test* - Efetua o procedimento de teste em uma rede neural.

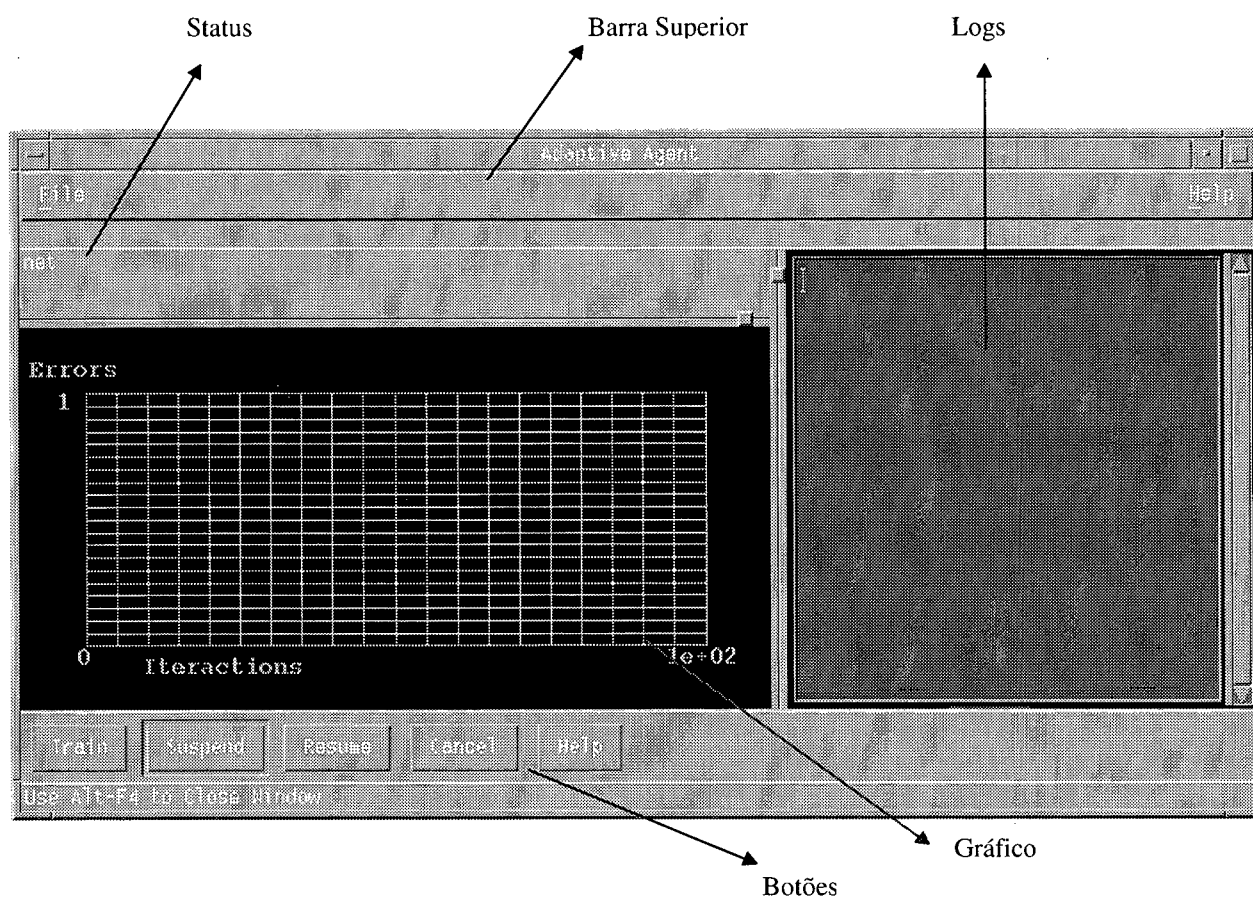


Figura 9-1 Interface do NeuroProto

9.1.3 Janela Gráfica

A janela gráfica mostra a evolução do processo de treinamento. Nesta janela, com o decorrer do treinamento da rede neural, são exibidos os pontos correspondentes aos erros médios quadráticos (RMS) encontrados, em cada iteração.

No eixo vertical, o maior erro médio quadrático encontrado em cada iteração é exibido. Os valores válidos neste eixo variam entre 0 e 1.

O eixo horizontal representa o número de iterações acontecidas. O gráfico é auto ajustável, ou seja, com o decorrer do treinamento, se o limite de iterações do gráfico for ultrapassado, um novo limite maior que o anterior é calculado e o gráfico é reexibido com este novo limite.

9.1.4 Área de *Logs*

A área de *logs* mostra as mensagens que o processo de treinamento ou de teste emitem durante a sua execução.

As mensagens para o processo de treinamento podem ser:

- Informações descrevendo, em cada iteração, quais os pares de treinamento aprendidos (exibidos como um “.”), quais os não aprendidos (exibidos como um “X”) e a relação entre os dois.
- Término de treinamento, exibindo o tempo total decorrido no treinamento.
- Suspensão exibindo o tempo total de treinamento decorrido até o momento de suspensão.
- Erros e avisos emitidos pela execução normal do ambiente, como arquivo de definição não aberto, treinamento suspenso ou reasumido, etc.

Portanto, nesta área, o sistema avisa ao usuário qualquer anormalidade em alguma operação efetuada pelo ambiente.

9.1.5 Janela de *Status*

Informa o nome da rede sendo treinada e também indica o seu estado de execução o qual pode ser: treinada, suspensa ou não-treinada.

9.2 Processo de Definição de uma Rede Neural

O processo de definição de uma rede consiste em se definir os parâmetros para a construção da rede, tais como o número de neurônios da camada oculta, tolerância e taxa de aprendizagem.

Estes parâmetros são declarados no arquivo de definição, usando-se as palavras chaves definidas no quadro 9.1. O arquivo de definição deve ter o nome formado pelo nome da rede e a extensão “.def”, por exemplo, “net1.def”.

Parâmetro	Palavra Chave	Valor	Exemplo
Neurônios da Camada de Entrada	INPUT	inteiro	INPUT=3
Neurônios da Camada Oculta	OUTPUT	inteiro	OUTPUT=4
Neurônios da Camada de Saída	HIDDEN	inteiro	HIDDEN=5
Tolerância	TOLERANCE	valor entre 0 e 1	TOLERANCE=0.1
Taxa de Aprendizagem	LEARNRATE	valor entre 0 e 1	LEARNRATE=0.65
Momentum	MOMENTUM	valor entre 0 e 1	MOMENTUM=0.1
Pares de Vetores de Mínimos	MINVECS	vetor de valores reais	MINVECS=4 4 0.0, 0 0 0.0
Pares de Vetores de Máximos	MAXVECS	vetor de valores reais	MAXVECS=9.0 5.0 1.0, 1.0 1.0 1.0

Quadro 9-1 Parâmetros de Definição da Rede Neural

9.3 Processo de Treinamento

Depois de definido os parâmetros da rede, pode-se proceder com a fase de treinamento. Para isto, cria-se o arquivo que irá conter o conjunto de treinamento. Este arquivo deve ter o nome formado pelo nome da rede e extensão “.trn”.

Cada entrada do conjunto de treinamento consiste em um par de vetores de entrada e saída desejada correspondente, separados por vírgula. Cada um desses vetores deve conter elementos com valores contidos nos intervalos definidos pelos pares de vetores máximos e pares de vetores mínimos.

9.4 Processo de Teste

O processo de teste consiste na definição de um arquivo de teste contendo pares de valores que, não necessariamente, estão excluídos do arquivo de treinamento. Este arquivo deve ter o nome formado pelo nome da rede e a extensão “.tst”.

Desta forma, pode-se selecionar o botão de teste e obter, na saída da rede, a percentagem de acertos obtidos quando ela processou o arquivo de testes.

9.5 Exemplo de uma Prototipação

Com a construção das Bibliotecas e do Ambiente de Treinamento, será detalhada a prototipação de um agente adaptativo para controlar as linhas de um sistema de acesso remoto discado à redeUFSC, a rede local da Universidade Federal de Santa Catarina.

Este sistema monitora a utilização das linhas pelos usuários que são classificados em grupos. O agente, periodicamente, atua na alocação das linhas para cada grupo, verificando a sua utilização e tomando a decisão de aumentar ou diminuir as linhas dedicada a ele. Portanto o objetivo, é atingir a maior eficiência e performance possível na utilização das linhas.

9.5.1 Descrição do Domínio do Problema

O sistema de acesso remoto discado da UFSC é um sistema que disponibiliza, através de linhas telefônicas, o acesso remoto a toda redeUFSC e conseqüentemente, a toda rede Internet. Desta forma, o usuário deve possuir um modem, uma linha telefônica, um microcomputador e um programa de acesso para utilizar o sistema.

A figura 9.2 descreve os principais componentes do sistema. O roteador efetua autorização do usuário, estabelece o protocolo de enlace (ppp ou slip) e efetua o endereçamento IP.

A autorização de cada usuário é realizada com a ajuda de um servidor de autenticação. O servidor mantém um cadastro contendo uma entrada para cada usuário. Cada entrada de usuário contém um nome de conta e uma senha.

No momento da conexão, o usuário conectando-se em uma linha, informa a conta e a senha ao roteador que por sua vez, pede ao servidor de autenticação para validar a conta e senha. O servidor verifica no seu cadastro se a conta e a senha são válidos e devolve a resposta ao roteador que nega ou autoriza o acesso ao usuário de acordo com a resposta obtida.

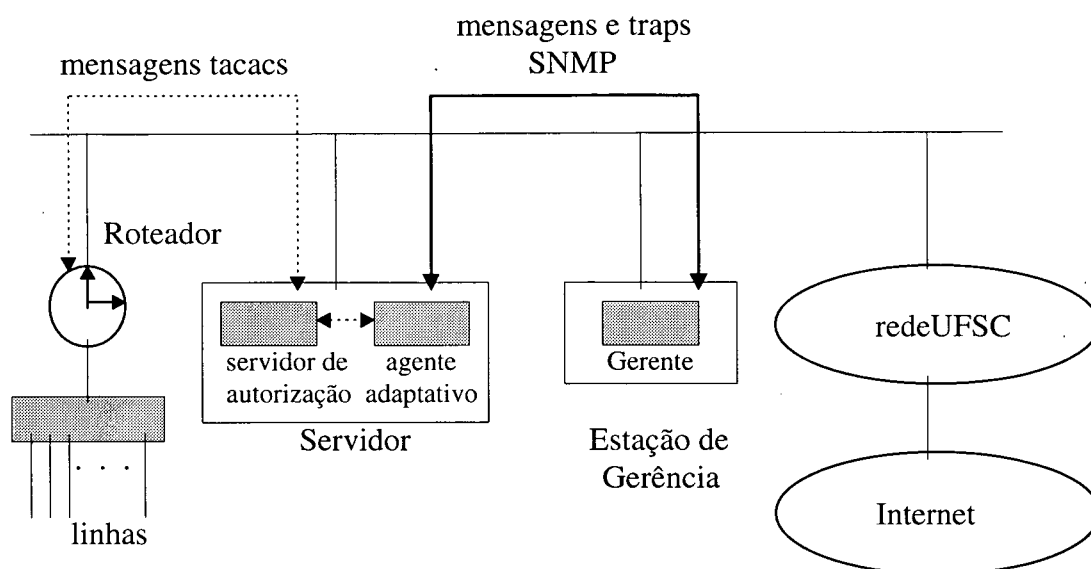


Figura 9-2 Componentes do Sistema de Acesso Remoto Discado

O funcionamento do sistema de acesso remoto como descrito considera a situação ideal, onde sempre o usuário, ao discar, encontra uma linha livre. Entretanto, devido às limitações de recursos, principalmente de linhas telefônicas disponíveis, o usuário pode não encontrar nenhuma disponível. Portanto, o conjunto das linhas telefônicas é um recurso que deve ser monitorado, procurando otimizar sua utilização pelos usuários.

9.5.2 Grupos de Usuários

Devido à necessidade de otimização do uso das linhas, decidiu-se dividir os usuários em grupos de acordo com sua prioridade de utilização, definida pela política de gerenciamento da redeUFSC.

Cada grupo, em um determinado momento, terá um número de linhas em que o acesso é garantido. Portanto, se num determinado momento, o grupo 1 tiver 15 linhas garantidas e somente 7 estão sendo usadas, no mínimo 8 usuários pertencentes ao grupo podem ter seu acesso permitido. Entretanto, se em outro momento, o grupo estiver com 15 linhas garantidas e as 15 linhas estão sendo usadas, então o acesso será negado para qualquer usuário daquele grupo.

Pode ser verificado na figura 9.2, que existe um componente, o qual é um agente adaptativo. Este agente periodicamente verifica a utilização das linhas e, atuando no servidor de autenticação, adiciona ou remove linhas de acesso para cada grupo de usuário. Desta forma

consegue-se controlar a utilização das linhas, procurando-se, de acordo como o treinamento do agente, um boa utilização das linhas.

Inicialmente serão testados um sistema com quatro grupos de usuários:

- Grupo 1 - agrupa os usuários que tem alta prioridade na utilização das linhas como os professores e funcionários.
- Grupo 2 - agrupa os usuários que tem uma prioridade de utilização mediana, tais como alunos de doutorado e mestrado.
- Grupo 3 - agrupa os usuários que tem uma prioridade de utilização também mediana, mas menor que o Grupo 2.
- Grupo 4 - agrupa os usuários que tem uma baixa prioridade de utilização como os alunos de graduação.

Serão definidos, por enquanto, somente estes três grupos de usuários. Entretanto a intenção é definir mais grupos, visando refinar o controle. Entretanto, esta possibilidade será indicada pelos resultados que forem obtidos da fase de prototipação.

9.5.3 Parâmetros de Controle

Para cada grupo de usuários, periodicamente são analisados três parâmetros pelo agente adaptativo:

- Número de tentativas de conexão sem sucesso pelos usuários do grupo, acumuladas nos últimos cinco minutos.
- Número de usuários conectados naquele momento.
- Número de linhas garantidas para o grupo de usuários.

Além disso, são considerados dois parâmetros do estado do sistema :

- Número de linhas utilizadas,
- E a hora da medição do número de linhas utilizado.

Portanto, obtendo-se os parâmetros acima para todos os grupos, o agente determina se algum grupo deve ter o seu número de linhas garantidas aumentado, diminuído ou deixado sem alteração, visando encontrar uma utilização das linhas melhor da atual.

9.5.4 Treinamento da Rede Neural

- **O processo de treinamento**

As entradas e saídas da rede neural consistem, basicamente, dos parâmetros de controle como foram definidos anteriormente. Cada vetor de entrada, será constituído por elementos que são os valores dos atributos de cada grupo, amostrados em períodos de tempo iguais. Cada vetor amostrado de entrada, será passado por uma fase de pré-processamento, sendo que cada elemento do vetor é pré-processado e colocado dentro de uma faixa de valores possíveis. Este vetor é, então, inserido na rede neural que produz como saída um outro vetor. A figura 9.3 resume melhor este processo.

- **Entrada e Saída da Fase de Pré-Processamento**

O pré-processamento consiste em transformar cada valor do vetor de entrada em um valor dentro de uma faixa de valores possíveis e com algum significado definido. Por exemplo, o primeiro elemento do vetor de entrada é a quantidade de pacotes transmitidos na interface, mas somente este valor não tem qualquer significado. Neste caso, o pré-processamento classifica se este tráfego é baixo, médio ou alto, que é o que realmente interessa para o agente adaptativo.

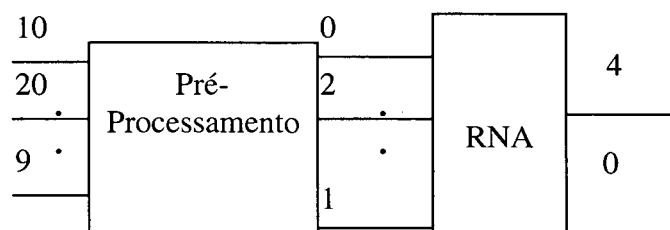


Figura 9-3 Pré-processamento

Para cada grupo, são obtidos o número de tentativas sem sucesso, o número de usuários conectados e o número de linhas garantidas. Como são quatro grupos, resultam em 12 parâmetros. Além disso dois parâmetros a mais são obtidos do sistema: a hora da amostragem e o número total de linhas conectadas. Os intervalos de valores resultantes do pré-processamento dos valores dos parâmetros de controle são dados pelo quadro 9.2.

Portanto, depois da fase de pré-processamento, tem-se um vetor de 14 elementos, onde cada seqüência de três valores representa os parâmetros de controle de um grupo.

Por exemplo, o vetor de entrada

$$[12 \ 3 \ 10 \ 0 \ 30 \ 20 \ 0 \ 25 \ 32 \ 0 \ 15 \ 32 \ 12 \ 8]$$

significa que no momento da amostragem, para cada grupo se tem:

- grupo 1: 12 tentativas sem sucesso, 3 usuários conectados naquele momento e 10 linhas garantidas.
- grupo 2: nenhuma tentativa sem sucesso, 30 usuários conectados naquele momento e 20 linhas garantidas.
- grupo 3: nenhuma tentativa sem sucesso, 25 usuários conectados naquele momento e 32 linhas garantidas.
- grupo 4: nenhuma tentativa sem sucesso, 15 usuários conectados naquele momento e 32 linhas garantidas

Sendo que a amostragem foi efetuada as 12 horas, tendo neste momento 8 linhas conectadas.

Depois da fase de pré-processamento, o seguinte vetor é originado:

$$[2 \ 0 \ 0 \ 0 \ 2 \ 2 \ 0 \ 1 \ 4 \ 0 \ 1 \ 4 \ 12 \ 0]$$

• Entrada e Saída da Rede Neural

A rede neural toma como entrada o vetor resultante da fase de pré-processamento e calcula um vetor de saída contendo dois elementos: o primeiro elemento informa o grupo que deve sofrer uma ação. O segundo elemento informa se a ação deve ser adição (1) ou diminuição (0) no número de linhas alocadas.

Portanto, o vetor

$$[2 \ 1]$$

indica que a garantia de linhas mínimas ao grupo 2 deve ser aumentada, ou seja, passada para uma garantia maior.

Os vetores especiais [0,0] indicam que o sistema não deve tomar ação alguma e os vetores [5,5] indicam que um alerta deve ser mandado ao gerente.

Tentativas sem Sucesso	<u>valor</u>	<u>intervalo</u>
	baixo(0)	0 - 3
	médio(1)	3 - 4
	alto(2)	> 4
Usuários Conectados	<u>valor</u>	<u>intervalo</u>
	baixo(0)	0 - 10
	médio(1)	10 - 25
	alto(2)	> 25
Hora da Amostragem Efetuada		<u>intervalo</u>
		0 - 24
Linhas Garantidas	<u>valor</u>	<u>garantia de linhas</u>
	0	10
	1	15
	2	20
	3	25
	4	32
	5	42
Linhas Conectadas	<u>valor</u>	<u>intervalo</u>
	0	0 - 10
	1	11 - 15
	2	16 - 20
	3	21 - 25
	4	26 - 32
	5	33 - 42

Quadro 9-2 Intervalos de Valores Resultantes do Pré-Processamento

9.5.5 Conjunto de Treinamento

Foram efetuados vários treinamentos com vários conjuntos de treinamento e feitos alguns testes. O quadro 9.3 resume os resultados obtidos:

Conjunto de Treinamento	Número de Pares de Treinamento	Taxa de Tolerância	Número de Iterações
A	54	0,07	3.234
B	64	0,07	10.753
C	64	0,05	99.250
D	96	0,07	não treinou

Quadro 9-3 Resultados Obtidos de Vários Conjuntos de Treinamento

A quadro 9.4 representa os testes, efetuados na rede treinada, com alguns pares de vetor de entrada e saída desejados e que não estavam no conjunto de treinamento.

Número de Pares para o Teste	Porcentagem de Acertos
10	90 %
30	95%
60	91 %

Quadro 9-4 Resultados do Procedimento de Teste

9.5.6 Construção do Subagente

A figura 9.4 ilustra o modelo de classe do agente adaptativo. Verifica-se na figura que o agente adaptativo foi modelado pela classe *monRAS*. Esta classe é derivada da classe *sAgent*, fornecendo a operação *run* que deixa o agente executando como um processo daemon.

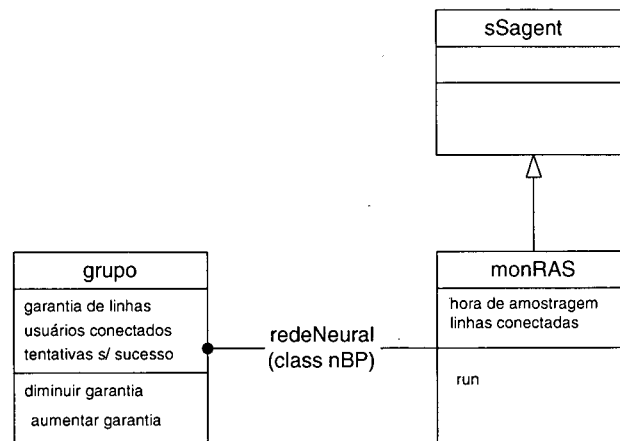


Figura 9-4 Modelo de classes do agente adaptativo

A classe grupo modela os grupos de usuários. Verifique que os atributos desta classe são justamente a garantia de linhas, o número de usuários conectados e as tentativas de conexão sem sucesso efetuadas. Além disso, esta classe contém duas operações: diminuir e aumentar a garantia, ambas, com significado óbvio.

Verifique que a rede neural (classe *nBP*) foi modelada como uma associação entre a classe *monRAS* e a classe grupo. Realmente, esta foi a melhor forma encontrada e também a mais elegante.

10. CONCLUSÃO

A construção de agentes adaptativos pode ser justificada para uma grande variedade de problemas. Neste trabalho, foi apresentada uma das maneiras de se construir agentes adaptativos utilizando redes neurais.

Como pode ser verificado, o objetivo principal do trabalho é construir um ambiente de desenvolvimento que permitisse a construção de agentes adaptativos e, para isto, foi empregada uma metodologia de desenvolvimento orientada a objetos. O produto final desta análise foi o desenvolvimento de uma Biblioteca de Classes de Redes Neurais e de Gerência de Redes.

Por fim, foi desenvolvido um programa de prototipação e treinamento de redes neurais. Este programa deve ser utilizado para validar a utilização de uma rede neural dentro do domínio de um determinado problema. Além disso, pode-se testar várias configurações de redes neurais visando encontrar a melhor possível.

A apresentação do ambiente de desenvolvimento pode ser resumida em vários aspectos: a integração das tecnologias de suporte, a utilização e o desenvolvimento das bibliotecas de classes e o emprego do programa de prototipação e treinamento. Cada um destes aspectos serão revistos e comentados sucintamente.

10.1 Integração das Tecnologias de Suporte

A construção de uma agente adaptativo foi apresentada , integrando-se três áreas tecnológicas:

- Gerência de Redes de Computadores
- Redes Neurais Artificiais
- Análise e Projeto Orientado a Objetos

O agente adaptativo foi apresentado como uma extensão ao conceito de agente dentro do modelo convencional agente-gerente. O trabalho foi totalmente direcionado para ser aplicado

dentro do padrão *Internet*, justificado pelo seu largo emprego comercial e sua extrema simplicidade.

A proposta principal foi estender um agente SNMP, através do conceito de subagente. Além disso, ao subagente foi adicionado características adaptativas, através do uso de uma rede neural atuando no sentido de controlar os objetos gerenciáveis associados. Desta forma, a rede neural deve reagir aos problemas e falhas apresentados pelos objetos gerenciáveis tentando manter a rede em condições normais de funcionamento.

Portanto, o agente adaptativo integra duas tecnologias:

- A Gerência de Redes de Computadores que fornece todo o suporte de comunicação agente-gerente através do protocolo SNMP e a estrutura de informação de gerenciamento com o emprego da SMI da Internet; e
- Redes Neurais que implementam a característica adaptativa através do emprego de vários tipos de redes neurais.

Com estas tecnologias, foi empregada uma metodologia orientada a objetos denominada OMT - *Object Modeling Technique*, para que determinados requisitos iniciais fossem atingidos. O emprego de análise e projeto orientado a objetos se mostrou extremamente valioso, apresentando as seguintes vantagens:

- O encapsulamento provido pelas classes, evitando que o usuário tenha que ter grandes conhecimentos sobre gerência de redes ou de redes neurais;
- A modularidade, ou seja, o código resultado do agente adaptativo é composto de uma série de módulos na forma de classes de objetos. O acesso a cada classe é realizado através de sua interface pública. Isto facilita a compreensão e prove uma boa estruturação do código;
- A estensibilidade provida pelas classes. Isto significa que uma classe pode ser facilmente estendida através do mecanismo de herança, o que possibilita a construção de classes mais especializadas sem a necessidade de reescrever todo o código;
- A uniformidade de conceitos usados, tanto na fase de análise, quanto na fase de projeto, através do emprego de classes com atributos e operações; e
- Em particular, a metodologia OMT modela um sistema usando três modelos: de classes, dinâmico e funcional. Cada um destes modelos descreve os três aspectos de um sistema: estático, comportamental e funcional.

10.2 Desenvolvimento das Classes Básicas e do Programa de Prototipação e Treinamento

Foram desenvolvidas duas bibliotecas de classes: Gerência de Redes e Redes Neurais.

A Biblioteca de Classes de Gerência de Redes é a mais extensa e mais complexa neste trabalho. Isto se deve em grande parte pela necessidade de prover todo o mecanismo de comunicação SNMP entre o subagente e o agente e também, todo o suporte à SMI da Internet. Portanto, foram definidas classes que podem ser derivadas ou usadas diretamente para instanciar objetos que possibilitem os registros de tipos de objetos (objetos gerenciáveis, segundo o termo da Internet), o recebimento e envio de PDUs contendo mensagens e *traps* SNMP e a estruturação de objetos gerenciáveis.

A Biblioteca de Classes de Redes Neurais é pequena e no momento somente traz implementada uma classe de rede neural com treinamento *Backpropagation*. Entretanto, esta biblioteca deve ser estendida para suportar mais classes de redes neurais, tanto redes com treinamento supervisionado, como redes com treinamento não supervisionado. Para isto, foi definido um modelo de classes OMT que deve ser seguido por estas futuras extensões.

Todas as duas bibliotecas tem suas classes modeladas segundo a metodologia OMT. Portanto, para as duas bibliotecas, diagramas de classes, diagramas de estados e diagramas de fluxo de dados foram definidos. Toda a implementação foi feita tendo como base os diagramas de classes, utilizando-se o C++ como linguagem de programação.

O programa de prototipação e treinamento tem as seguintes funções no desenvolvimento de um agente adaptativo:

1. Validação da solução, via redes neurais, visto que nem todos os problemas pode ser resolvidos eficientemente com emprego de redes neurais, sendo que muitas vezes a utilização de outras técnicas pode se mostrar mais adequada.
2. Prototipação das redes neurais, encontrando as configurações mais adequadas para o problema.
3. Realização do treinamento, ou seja, através do programa se constroem as sinapses da rede neural que mais tarde serão utilizadas pelo agente adaptativo na sua execução. Portanto, o agente adaptativo não necessita ter a capacidade de treinamento.

O programa de prototipação e treinamento é executado no sistema operacional UNIX, utilizando o ambiente gráfico *Common Desktop Environment/Motif 1.2*. As Bibliotecas de Classes de Gerência de Redes, para serem utilizadas, necessitam que os agentes SNMP suportem o

protocolo SMUX, ou que em sua maioria, sejam agentes implementados em ambientes UNIX. A Biblioteca de Redes Neurais inicialmente não tem um requisito e pode ser facilmente portada para outros sistemas operacionais diferentes do UNIX.

10.3 Visão de Usuário

O desenvolvedor ou implementador de um agente adaptativo deve seguir os seguintes passos no seu desenvolvimento:

1. Validar a solução, encontrar a melhor configuração e construir as sinapses da rede neural. Isto é feito usando-se o programa de prototipação e treinamento. Um ou vários conjuntos de treinamento devem ser formados pelo usuário. Em seguida, a rede deve ser treinada para cada conjunto de treinamento, onde se testa sua convergência dentro de várias configurações possíveis. Encontrada a melhor configuração, as sinapses da rede são salvas.
2. O implementador deve construir classes derivadas da classe de objeto gerenciável (*sMO* na *SNMPLib*) para acomodar as características inerentes dos objetos gerenciáveis que o subagente irá controlar. Além disso, os objetos podem ter características adaptativas implementadas através de redes neurais. Estas características podem ser modeladas como:
 - Uma nova classe de objeto gerenciável, derivada de uma classe de rede neural.
 - Uma relação de agregação com uma classe de rede neural, ou seja, a rede neural é definida como um atributo de um objeto gerenciável.
 - A rede neural é uma relação de associação entre as classes de objetos gerenciáveis.
3. Definir uma classe derivada da classe *sAgent* para implementar o agente adaptativo que monitorará os objetos gerenciáveis. Este agente irá conter outras características particulares ao domínio do problema.
4. Com o agente adaptativo construído, coloca-se o mesmo para executar como um *daemon* que monitora os objetos gerenciáveis.

No caso em que o agente adaptativo e os objetos gerenciáveis tem pouca complexidade, a implementação pode ser feita com poucas linhas de código. Portanto, a velocidade de

desenvolvimento de agentes adaptativos e não adaptativos é aumentada. Além disso, o implementador fica livre para preocupar-se com as peculiaridades inerentes à implementação, principalmente com respeito aos objetos gerenciáveis.

10.4 Alcance dos Objetivos

O objetivo geral foi construir o suporte necessário para o desenvolvimento de agentes adaptativos utilizando redes neurais. Este objetivo foi amplamente alcançado. Mas com o desenvolvimento do trabalho, novos objetivos foram surgindo que devem ser considerados na análise final do trabalho:

- Através do uso de uma metodologia orientada a objetos, pode-se comprovar a sua extrema adequação e vantagens para o desenvolvimento do agente. A orientação a objetos não é necessário, mas recomenda-se fortemente o seu uso, principalmente de uma metodologia como a OMT.
- Foi construída uma biblioteca de classes de gerência de redes, a *SNMPLib*, que pode ser usada na construção de agentes convencionais, tornando, inclusive, rápida e eficiente a sua construção.
- Da forma como foi implementado, o princípio aberto-fechado foi amplamente conseguido, visto que cada classe pode ser redefinida, derivando-se uma nova classe. Além disso, e o que talvez seja mais importante, a documentação provida pela OMT torna fácil o entendimento de cada classe, permitindo sua manutenção com extrema rapidez. Portanto, a Biblioteca de Classes de Redes Neurais, a *Neurolib*, sendo a que mais necessita de extensões futuras, pode ser facilmente estendida, através de implementações de outros tipos de redes neurais.
- Através do uso do programa de prototipação e treinamento, *ProtoNeuro*, o desenvolvedor pode prototipar a rede neural e validar a solução, antes de começar a implementação. Realmente, isto é necessário, pois nem todo problema pode ser eficientemente resolvido com o uso de redes neurais.

Com estes objetivos atingidos, pode-se concluir que a construção de agentes adaptativos (ou não adaptativos) foi extremamente facilitada. Além disso, um método para efetuar tal

desenvolvimento pode ser considerado como o produto final do trabalho e talvez de significado mais relevante no seu contexto.

10.5 Dificuldades Encontradas

As maiores dificuldades ocorrem na construção da *SNMPLib*. A compreensão da SMI da Internet e dos protocolos SNMP e SMUX levou a construção de várias classes.

Entre estas classes, estão aquelas que definem a natureza sintática dos objetos gerenciáveis, derivadas da *classesOS*, e que foram inicialmente, difíceis de serem definidas.

Outra classe a que se deve algumas dificuldades foi a *sMO* usada para modelar um tipo de objeto, definido na Internet como objeto gerenciável. No primeiro momento da análise procurou-se definir a classe *sMO* equivalente ao termo usado na Internet. Mas depois de muitos problemas, foi notada a inadequação e a análise foi redirecionada.

As classes que auxiliam na formação e transmissão de uma PDU SNMP/SMUX exigiram também um esforço extra. Entre estas classes estão as classes que encapsulam as variáveis de uma PDU e a classe que implementa um tipo ASN.1, denominada *desTE*.

O desenvolvimento das classes da *Neurolib* foi o mais tranquilo, pois foi tomado como base o trabalho de Adam Blum[04]. Mas mesmo assim, foi necessário fazer uma nova análise dentro da metodologia OMT, o que levou a muitas modificações e extensões. Pode-se dizer que desta análise, pouco código restou do trabalho original de Adam Blum.

Por fim, o desenvolvimento do programa de prototipação e treinamento, *NeuroProto*, foi facilitado pela utilização das classes da *Neurolib*, sendo necessário esforço extra para o estudo do ambiente CDE/Motif, o que levou algum tempo. Deste esforço de implementação, resultou outro conjunto de classes que podem ser usadas para construção de interfaces gráficas no CDE. Entretanto, não foram citadas por fugirem ao escopo do trabalho.

10.6 Aplicabilidade de Agentes Adaptativos

O uso de agentes adaptativos é recomendável quando necessita-se de uma gerência ia automatizada ou pró-ativa. Neste sentido, o agente adaptativo age como um gerente localizado

no nó gerenciável, evitando possíveis intervenções de um operador humano para reagir aos problemas ou falhas na rede.

Entretanto, o uso de agentes adaptativos com a utilização de rede neurais é mais indicada onde a precisão das decisões não é um requisito muito forte. Mesmo assim, a quantidade de aplicações possíveis são enormes. Como exemplo, foi descrita a implementação um agente adaptativo para controlar as linhas do sistema de acesso remoto discado implantado na redeUFSC.

Outra característica dos agentes adaptativos usando redes neurais é a rapidez para tomar uma decisão. De fato, somente o treinamento de um agente adaptativo é demorado. Uma vez treinado, o agente pode deduzir uma saída correspondente a uma entrada pois ele recorre às sinapses para lembrar o seu treinamento. Em geral, este processo é rápido, o que sugere a sua aplicabilidade no controle de certos processos de tempo real em redes de computadores como controle de tráfego em redes ATM ou controle de rotas em roteadores.

10.7 Perspectivas Futuras

O trabalho realizado, de forma alguma pode ser considerado concluído, apesar dos objetivos iniciais terem sido alcançados. Com a implementação em mãos, pode-se considerar novos objetivos, propondo uma continuidade ao trabalho:

1. Implementação e análise de novas classes de redes neurais, como classes com outros algoritmos de treinamento além do *Backpropagation* e classes com treinamento não-supervisionado.
2. Analisar e criar uma relação de problemas freqüentes que podem ser resolvidos eficientemente por redes neurais. Isto deve ser feito por especialistas em gerências de redes de computadores que devem passar para a rede neural, na fase de treinamento, o seu conhecimento prático. Um produto final, poderia ser a criação de uma biblioteca de objetos gerenciáveis adaptativos que poderia ser amplamente utilizada na comunidade Internet.
3. Análise o uso de algoritmos evolucionistas, como algoritmos genéticos para prover a características adaptativa aos objetos gerenciáveis.

Portanto, há ainda muita pesquisa e desenvolvimento a ser feito, entretanto, as bibliotecas de classes apresentadas já estão prontas para o uso imediato. Realmente, esta é uma das características do desenvolvimento orientado à objetos, o princípio aberto-fechado.

ANEXO A : DIAGRAMAS DA BIBLIOTECA SNMPLIB

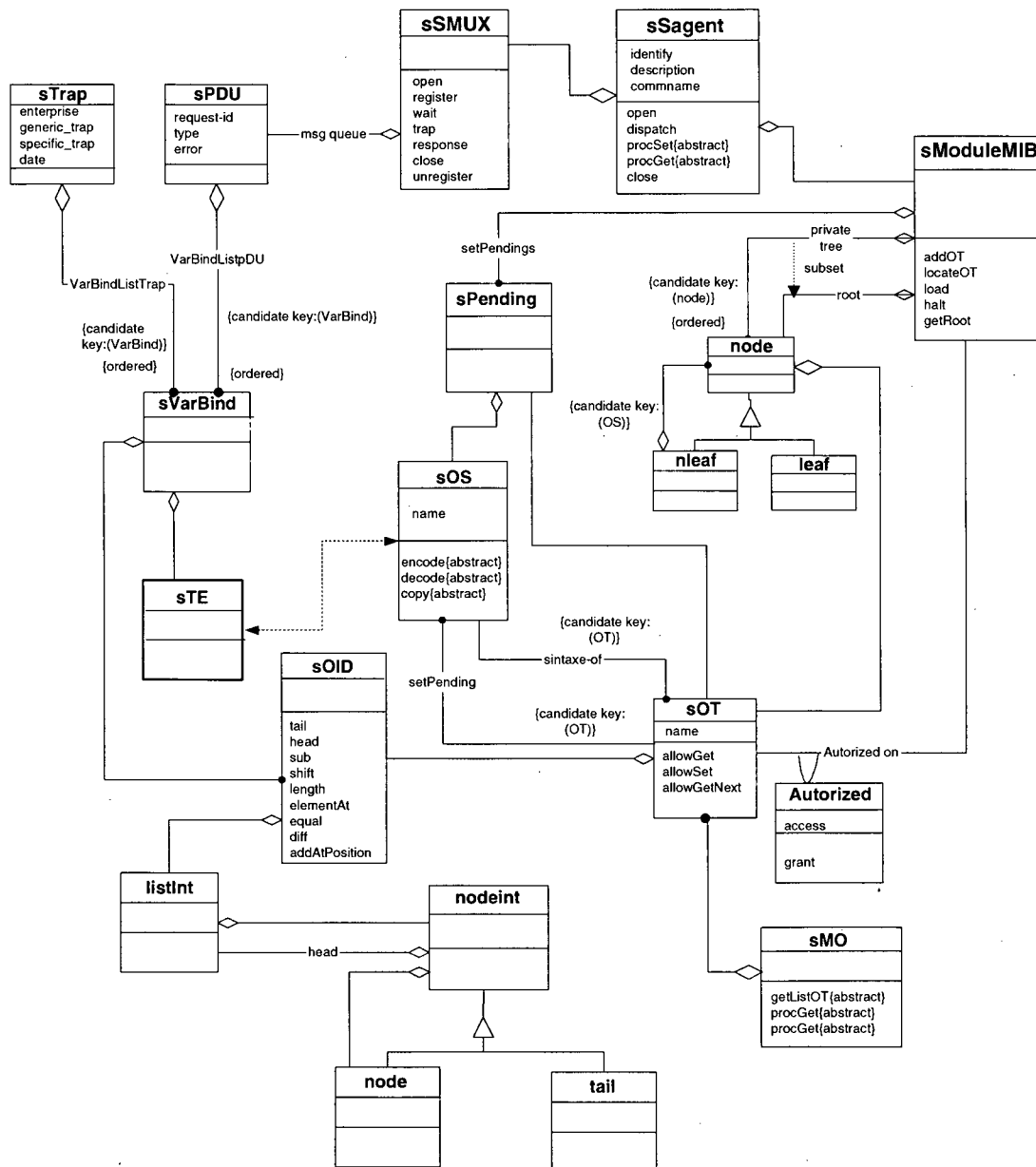
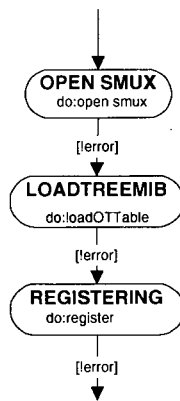


Diagrama de Classes da SNMPLib

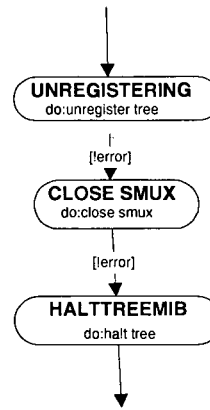
sSagent



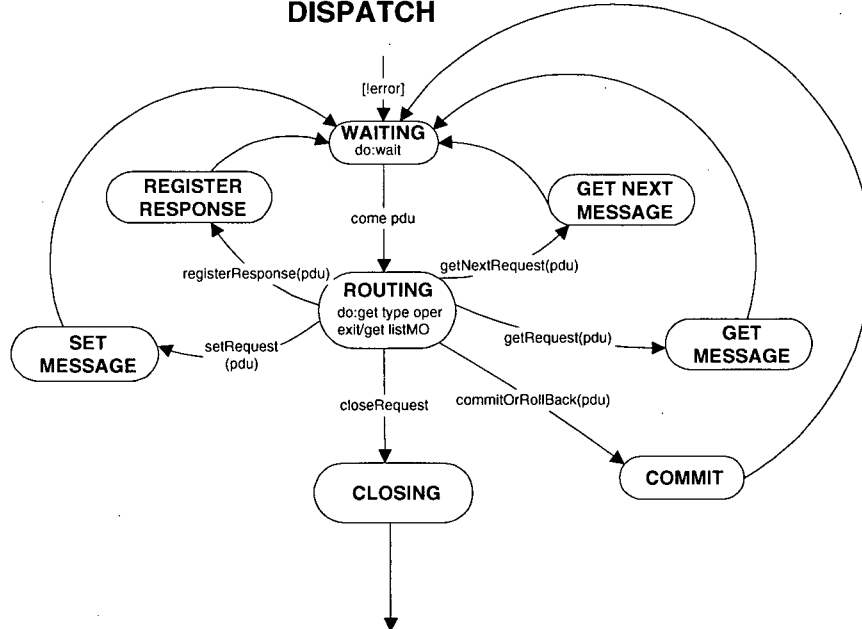
OPENNING



CLOSING

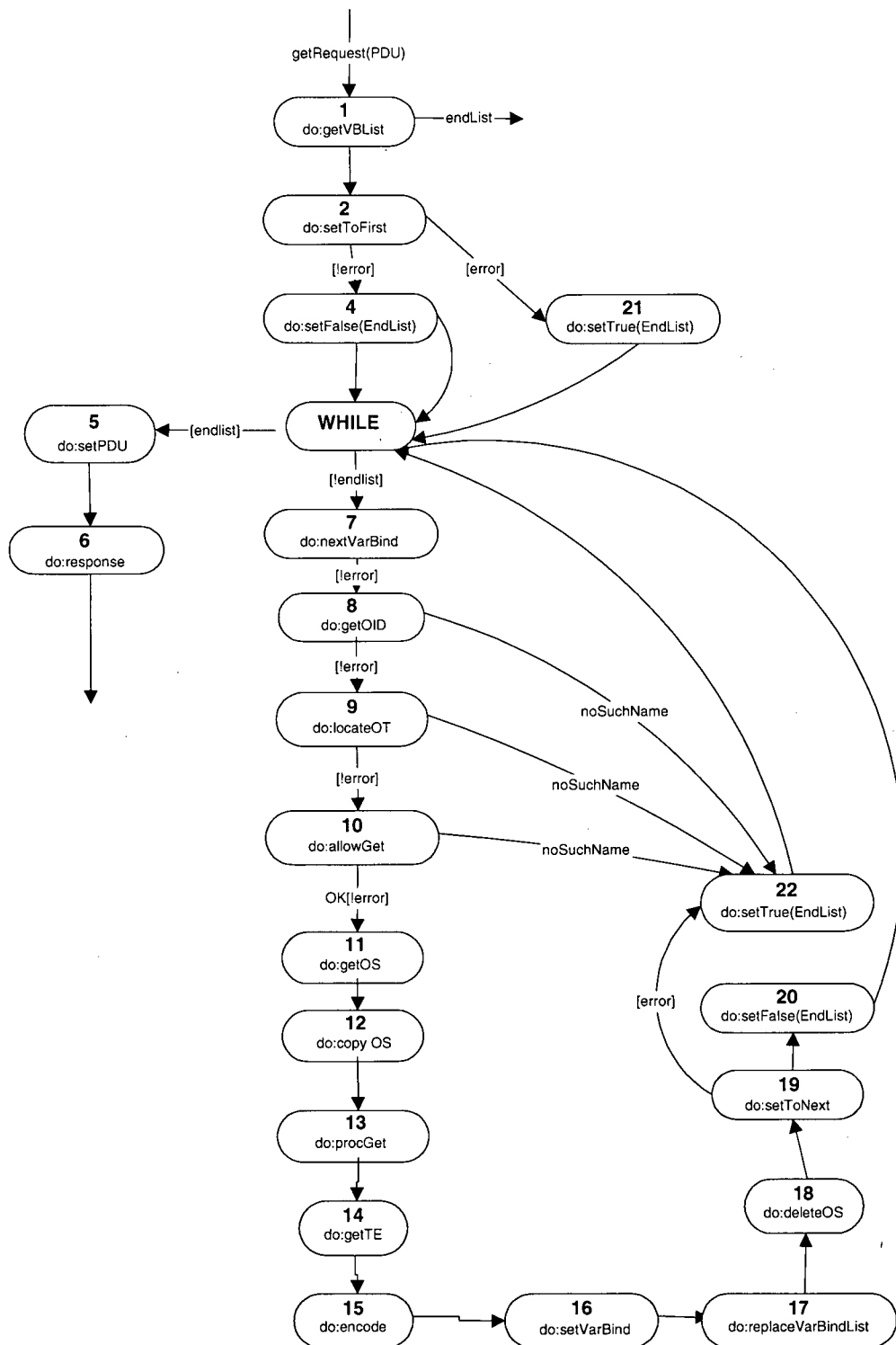


DISPATCH



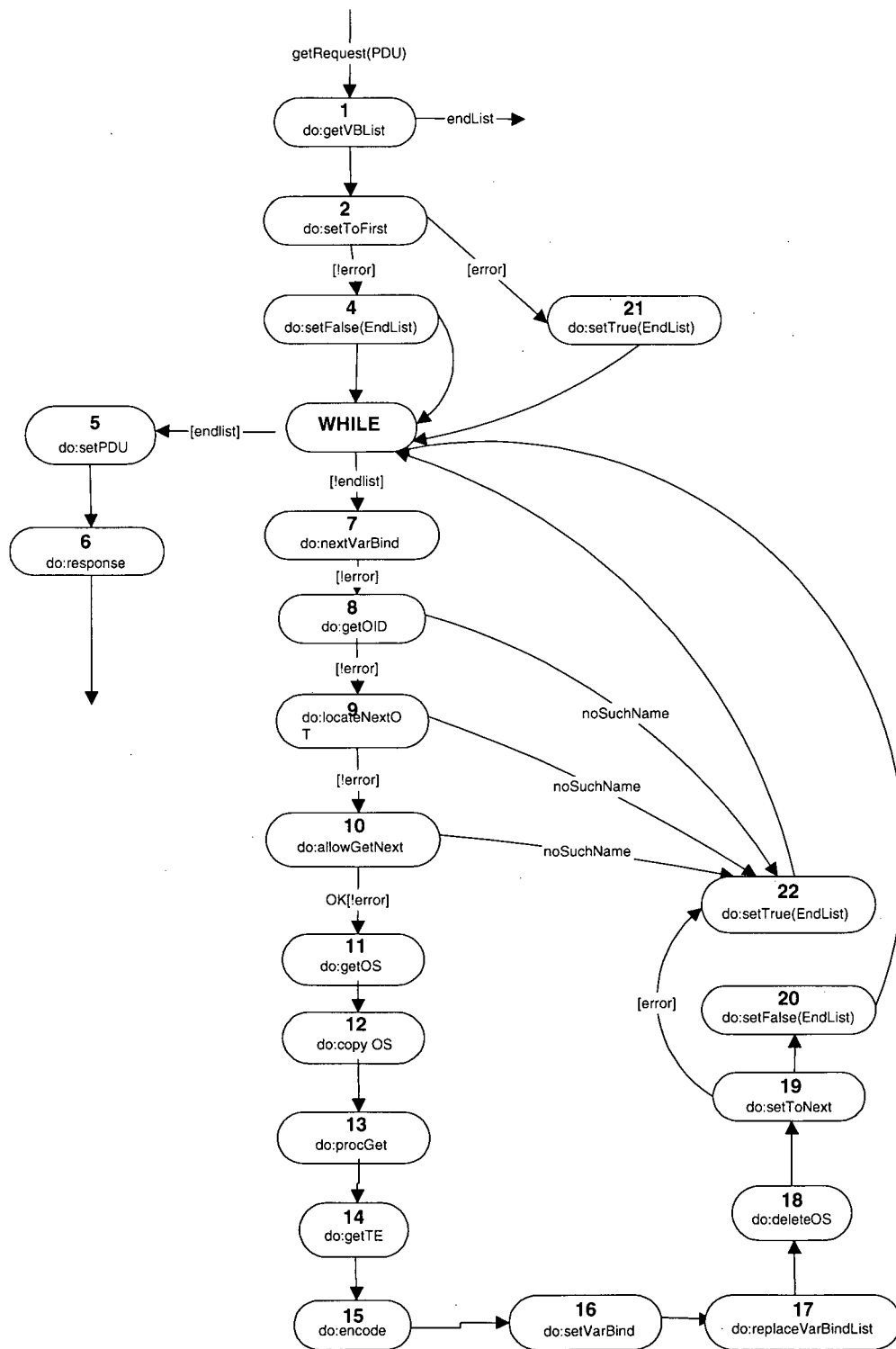
Diagramas de Estados - SNMPlib

GET MESSAGE



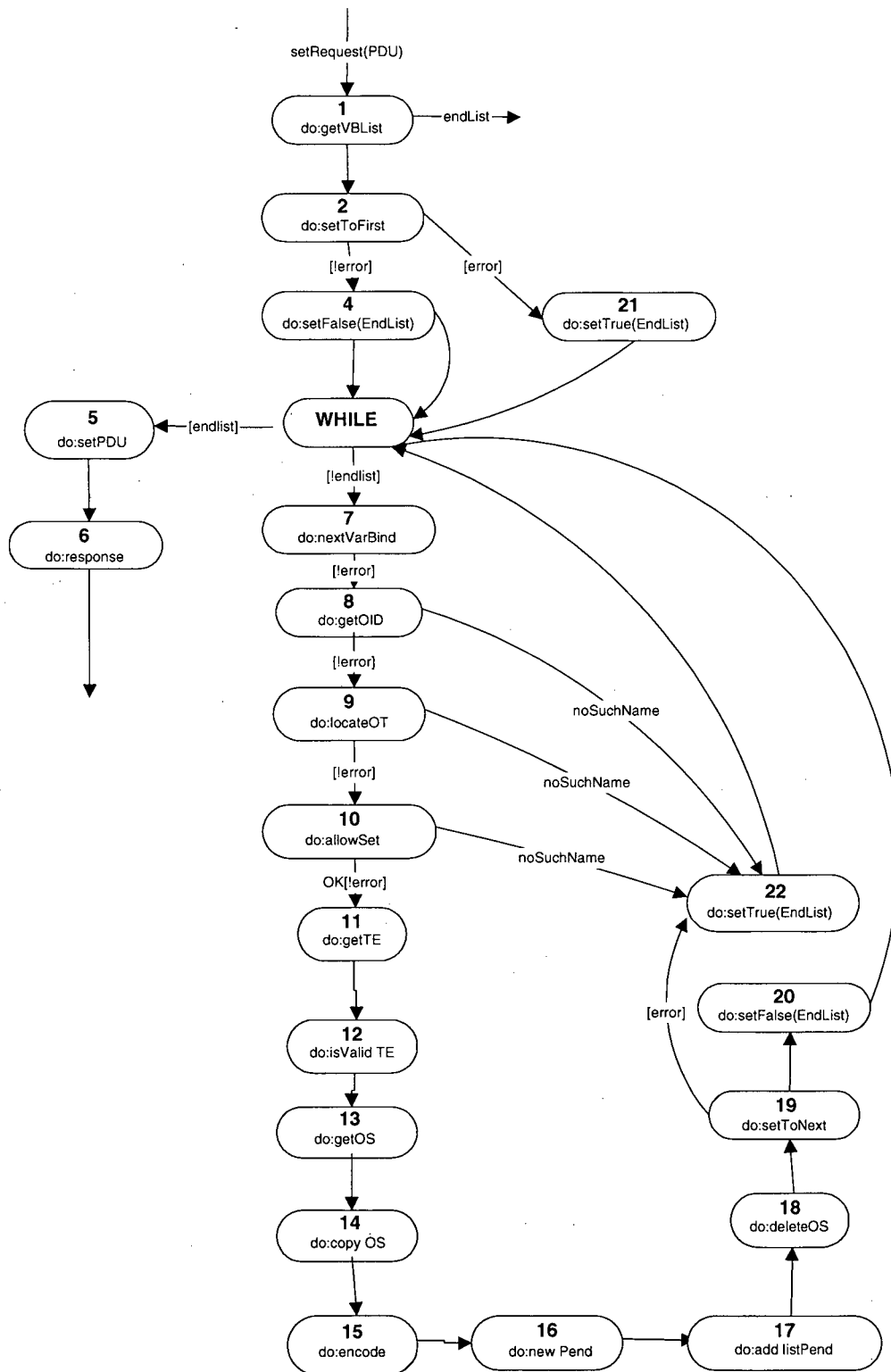
Diagramas de Estados - SNMPlib - Continuação

GETNEXT MESSAGE



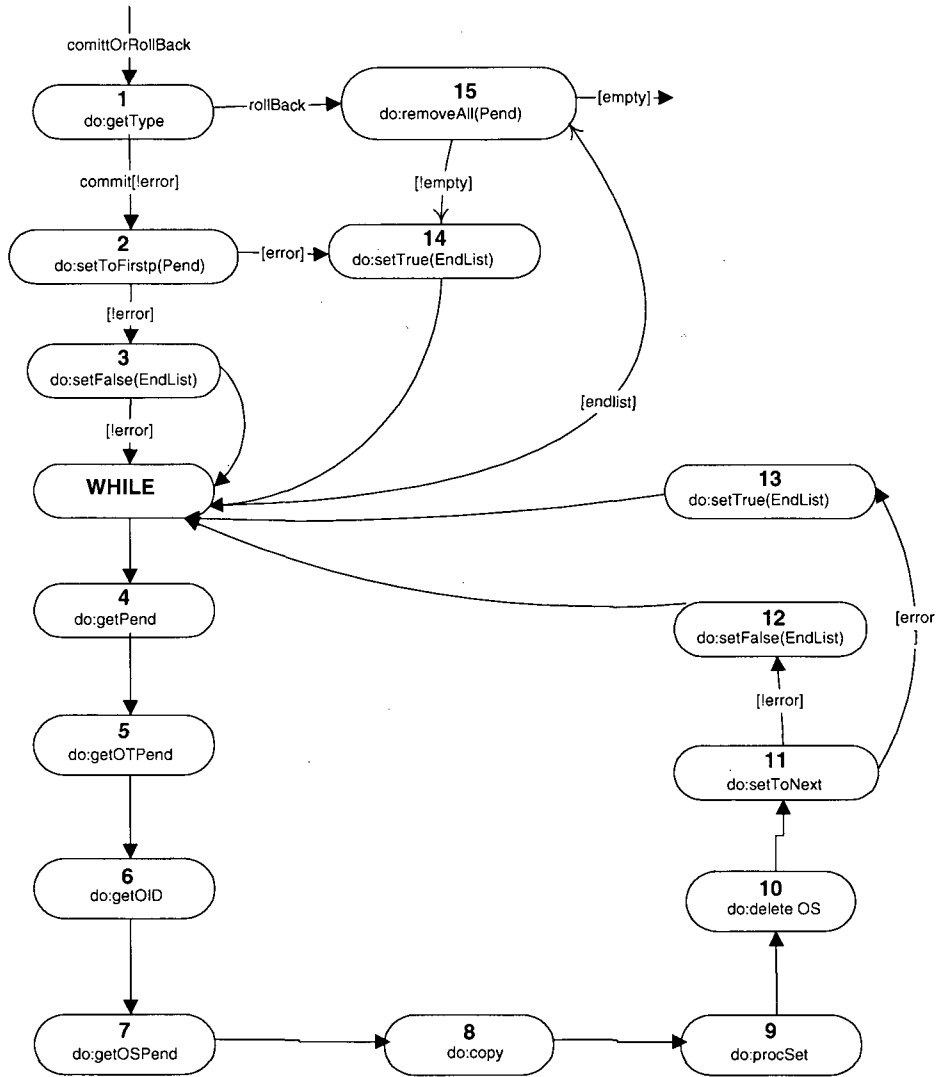
Diagramas de Estados - SNMPlib - Continuação

SET MESSAGE



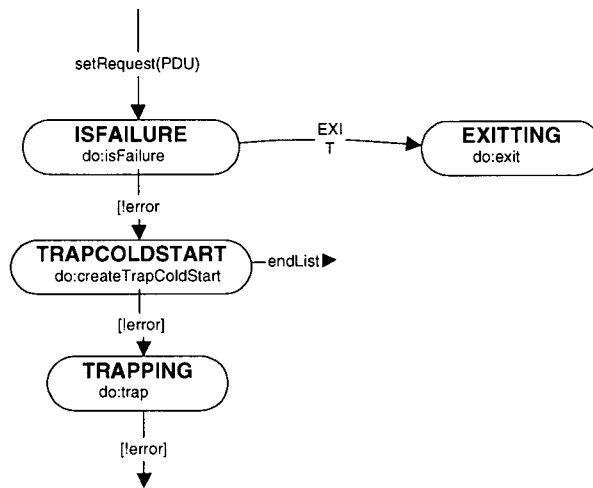
Diagramas de Estados - SNMPlib - Continuação

COMMITORROLLBACK



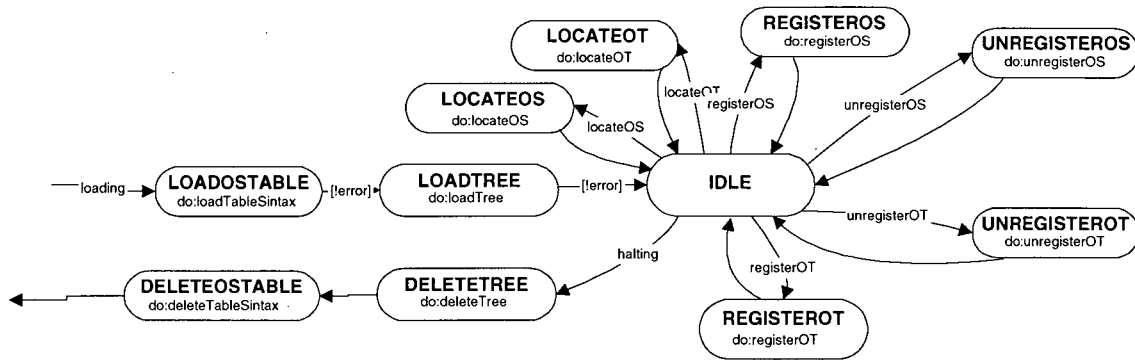
Diagramas de Estados - SNMPlib - Continuação

REGISTERRESPONSE

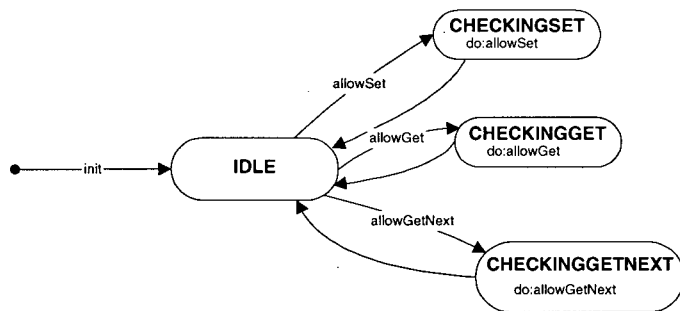


Diagramas de Estados - SNMPlib - Continuação

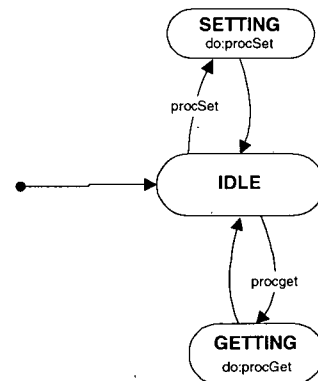
sModuleMIB



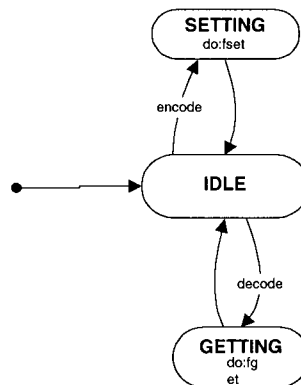
sOT



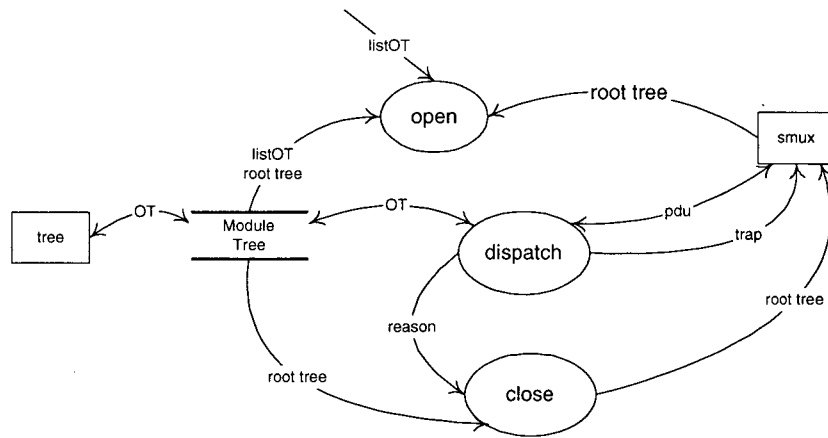
sMO



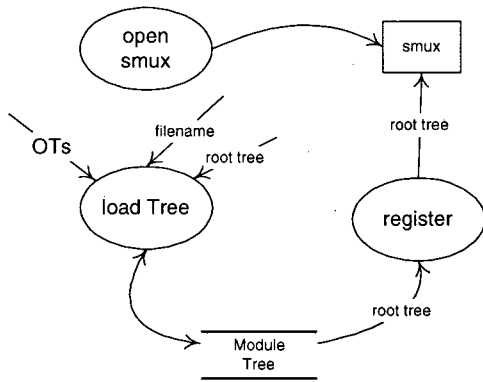
sOS



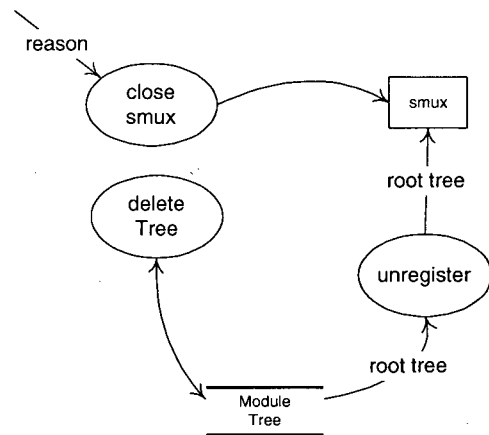
sSagent



Processo Open

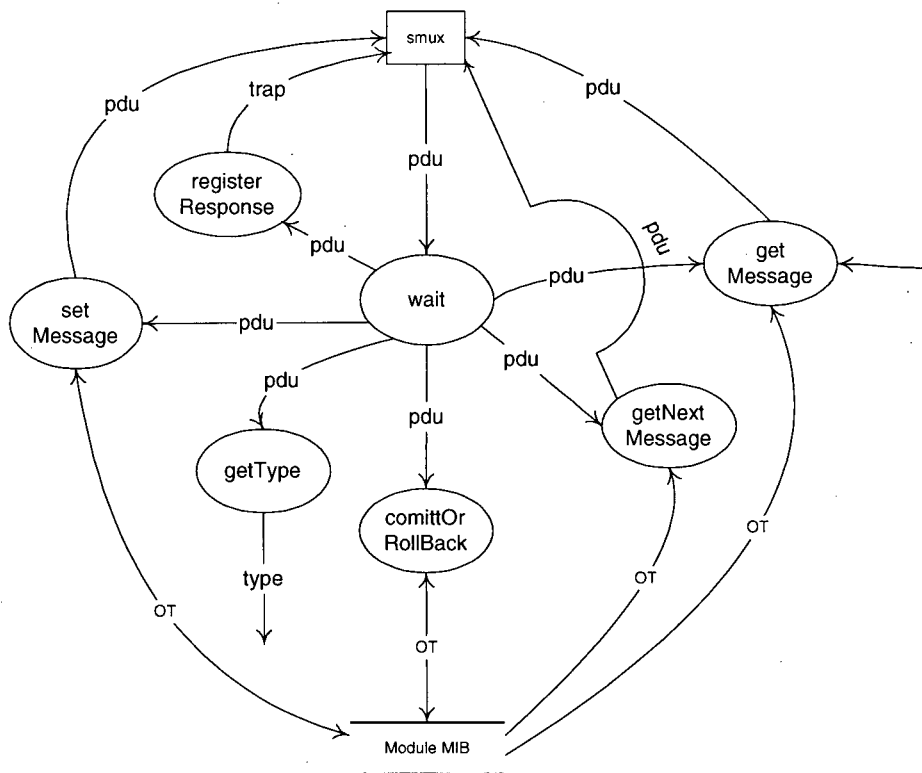


Processo Close

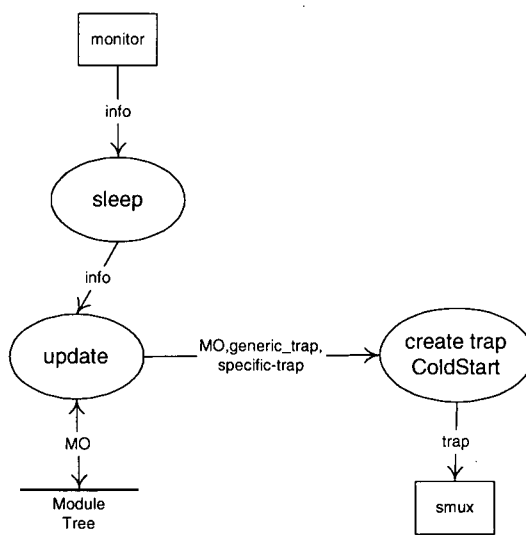


Diagramas de Fluxo de Dados - SNMPlib

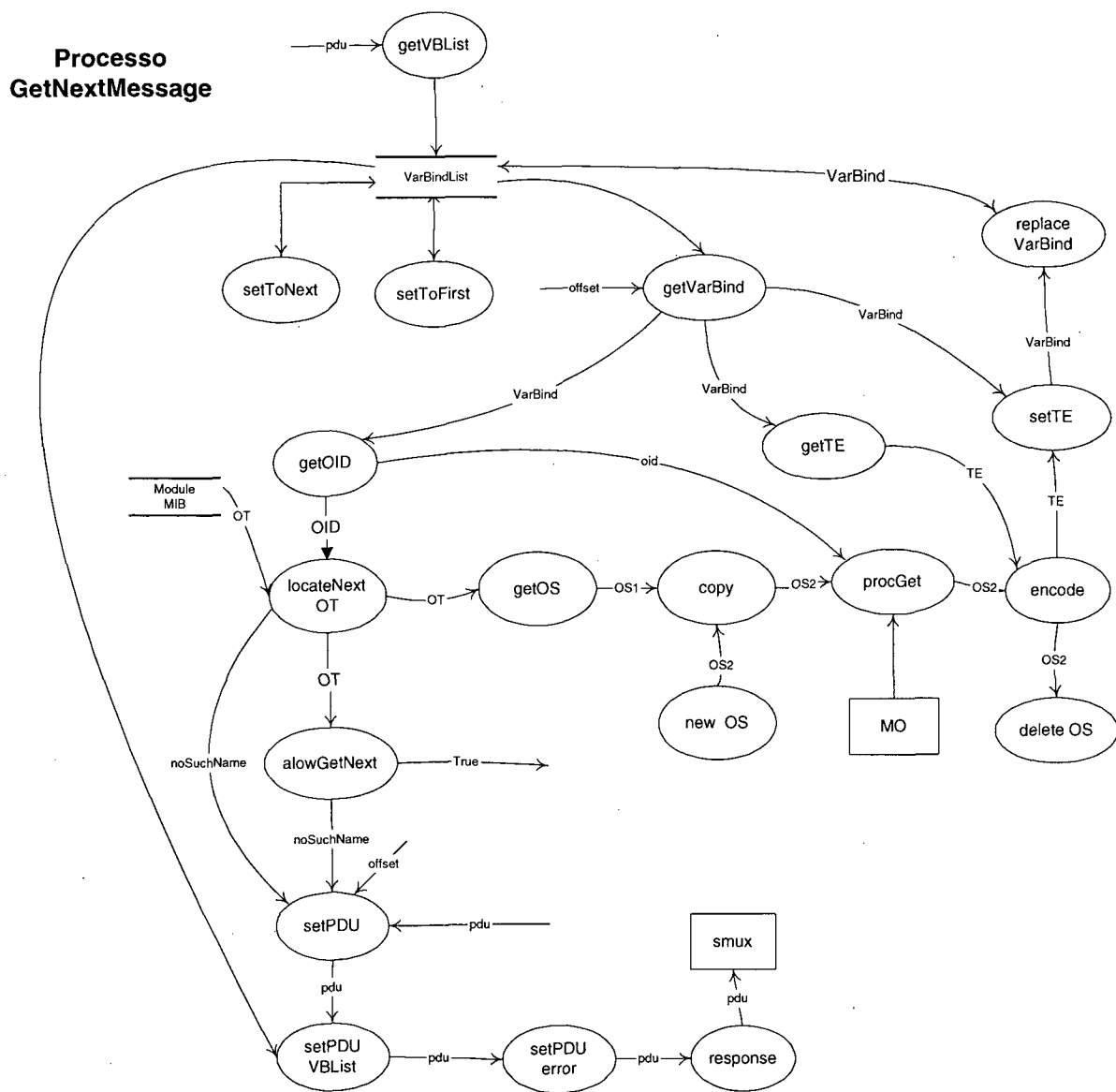
Processo Dispatch



Processo Control

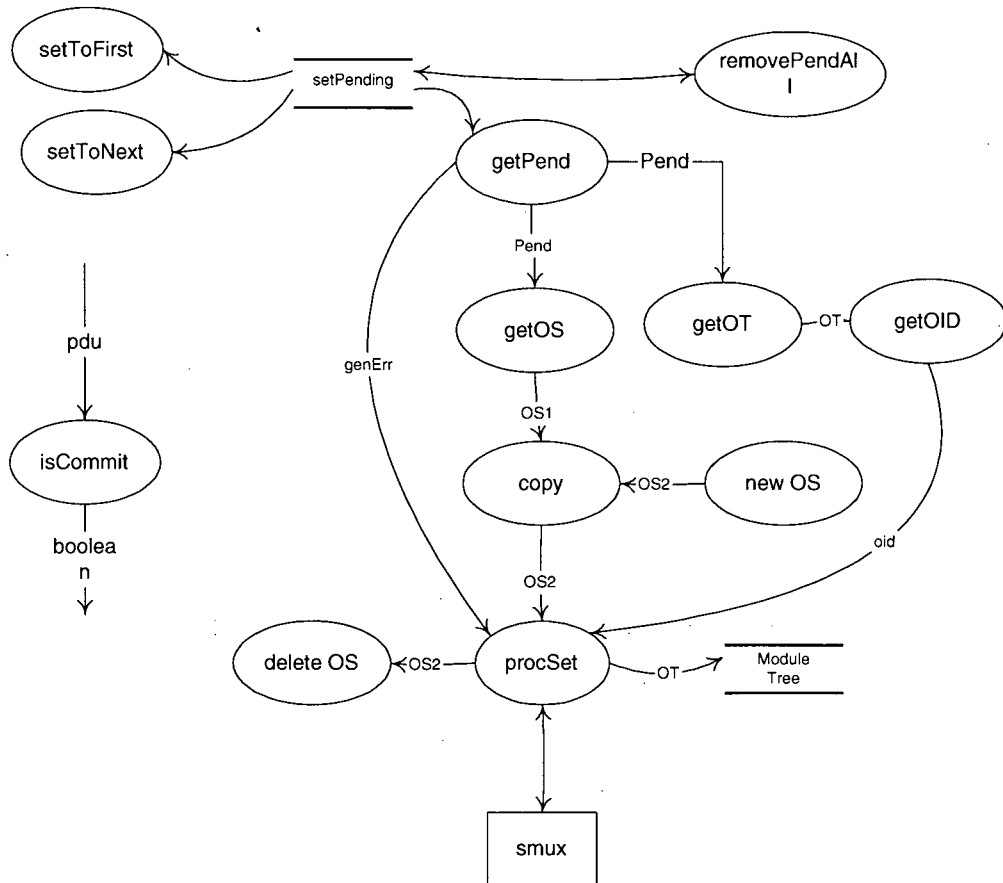


Diagramas de Fluxo de Dados - SNMPlib - Continuação

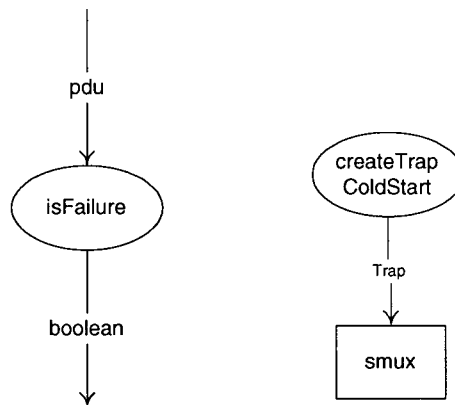


Diagramas de Fluxo de Dados - SNMPlib - Continuação

Processo CommitOrRollBack

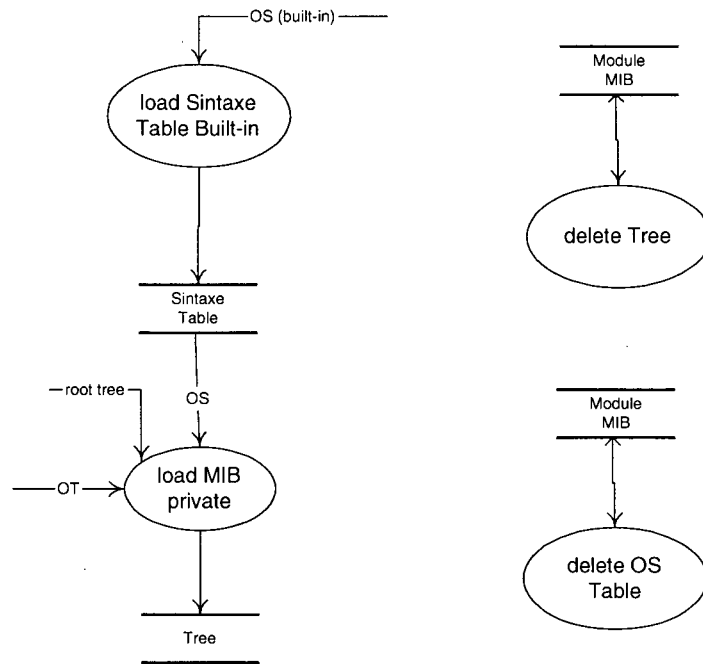


Processo RegisterResponse



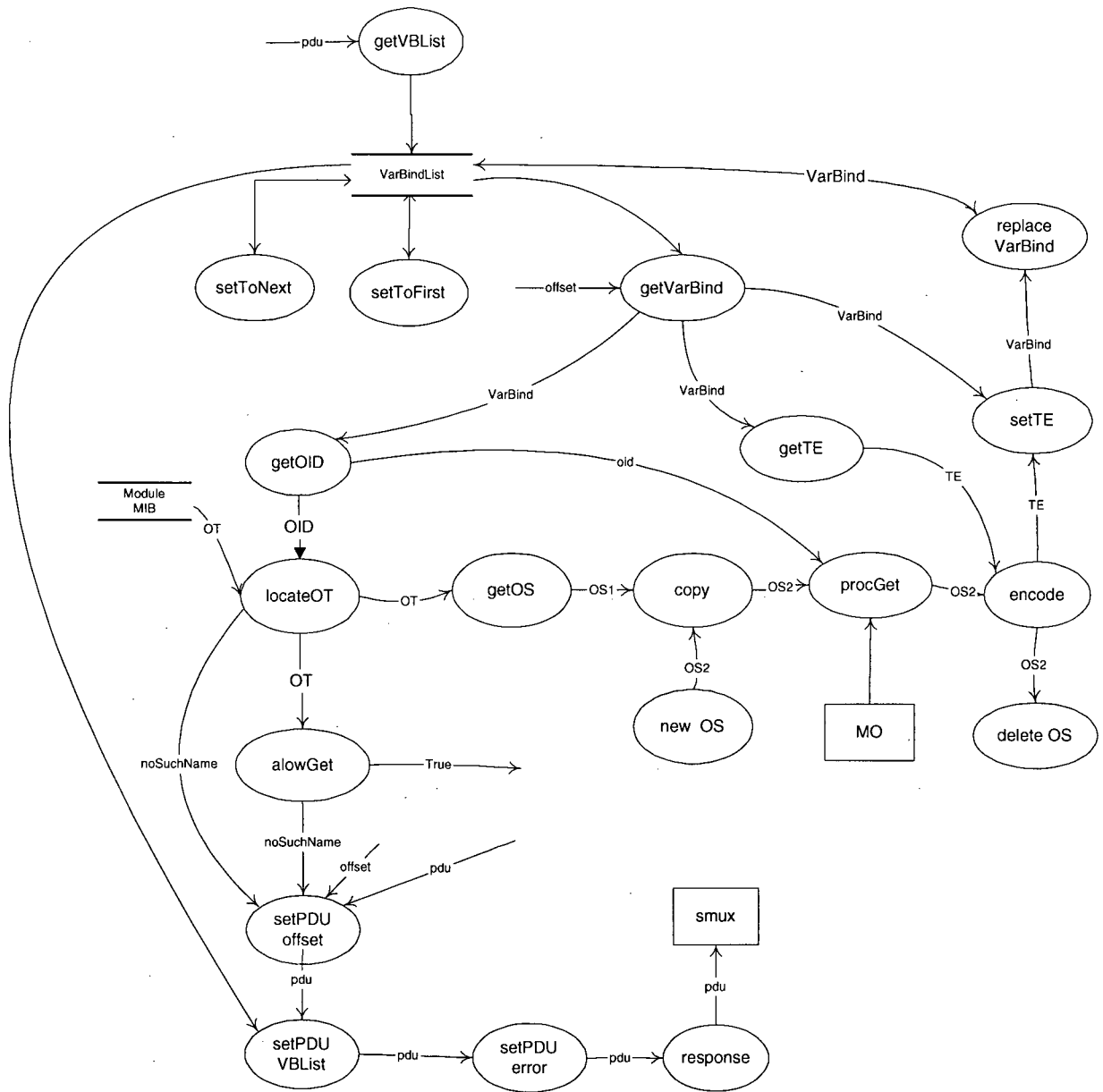
Diagramas de Fluxo de Dados - SNMPlib - Continuação

Processo LoadMIB



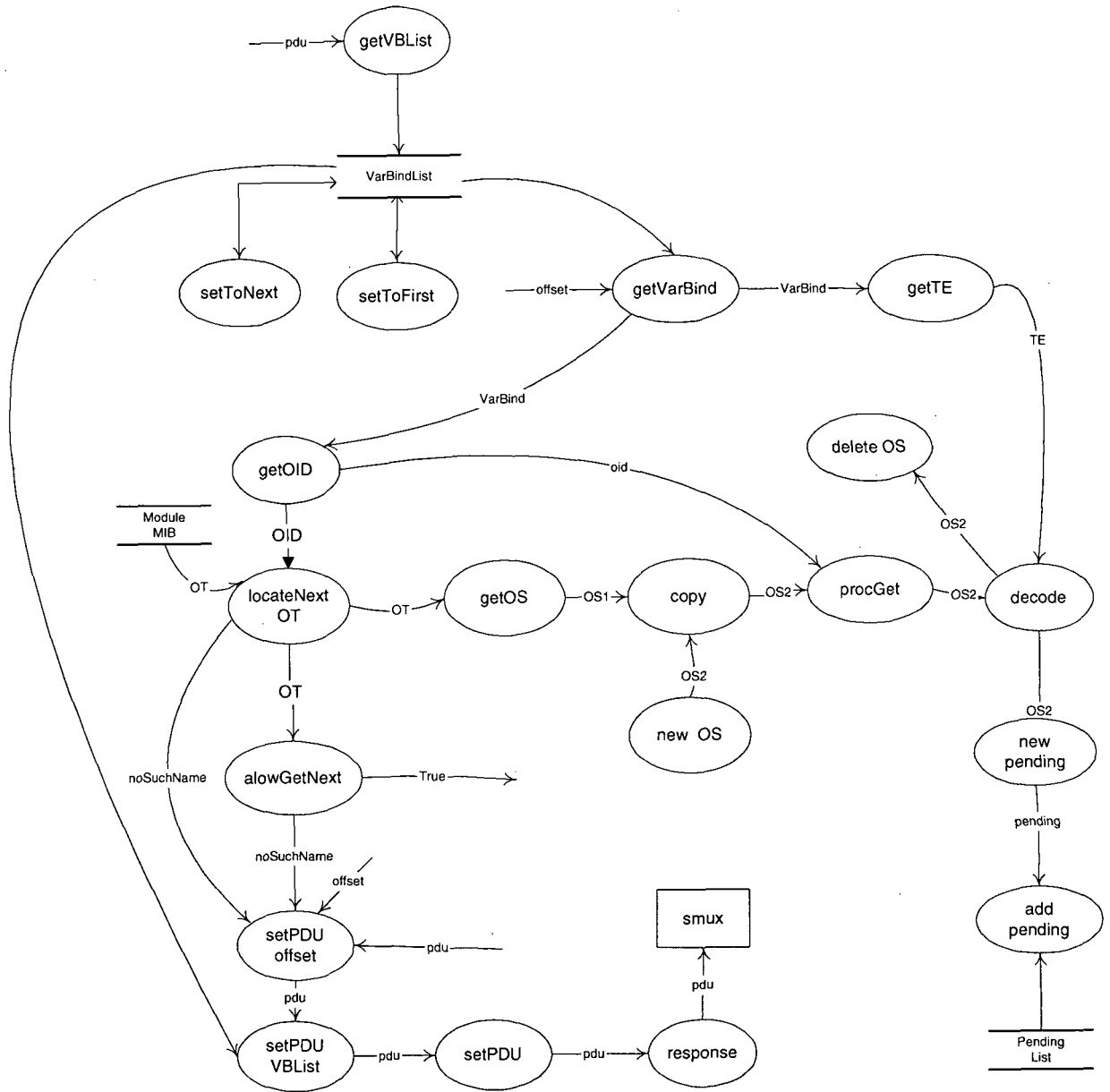
Diagramas de Fluxo de Dados - SNMPLib - Continuação

Processo getMessage



Diagramas de Fluxo de Dados - SNMPlib - Continuação

Processo SetMessage



Diagramas de Fluxo de Dados - SNMPlib - Continuação

ANEXO B: DIAGRAMAS DA BLIBLIOTECA NEUROLIB

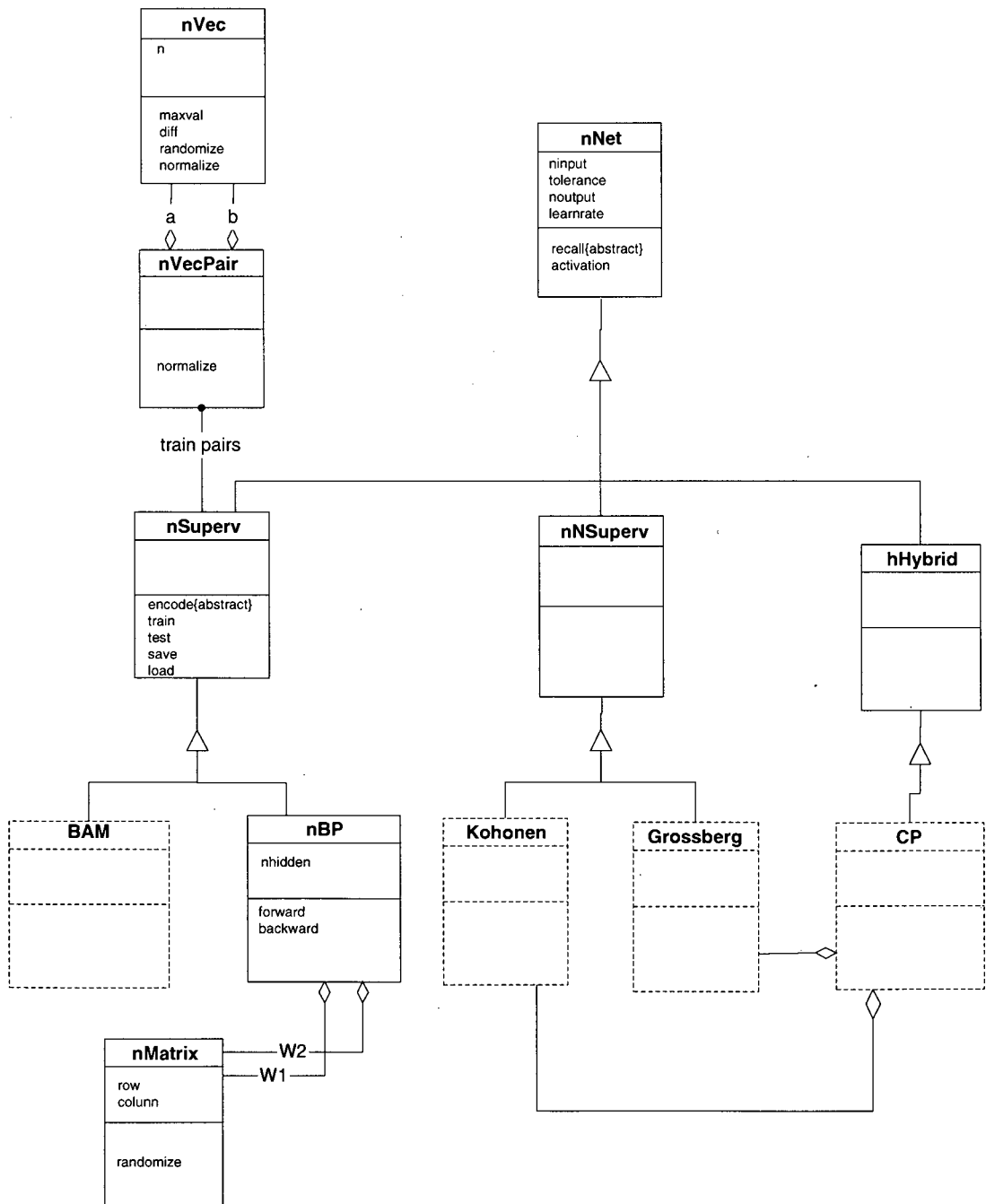


Diagrama de Classes - Neurolib

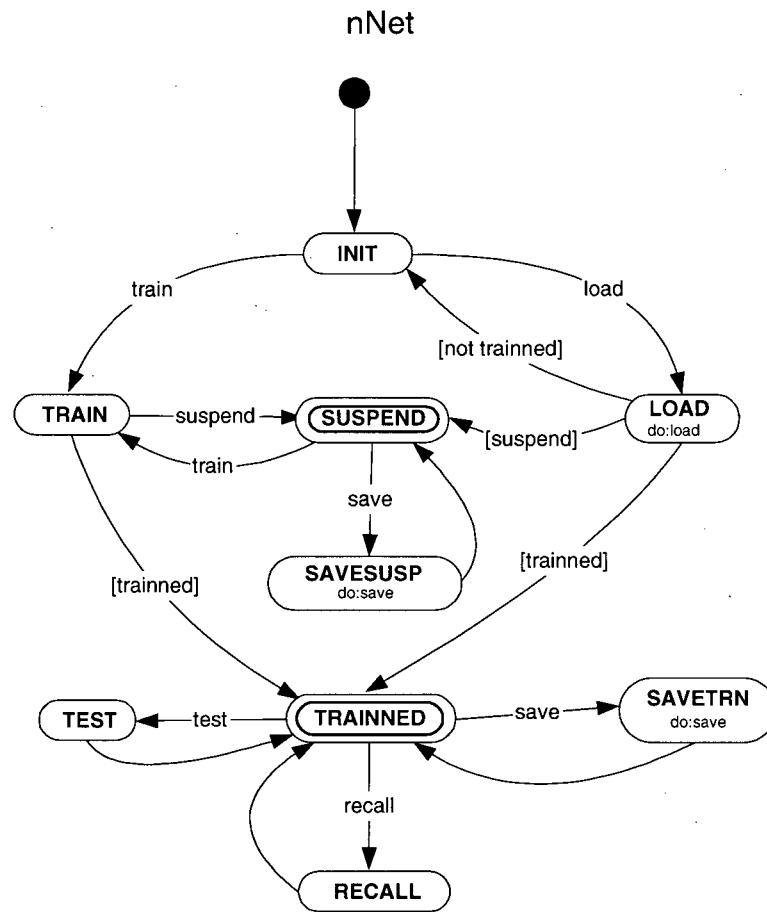


Diagrama de Estados - Neurolib

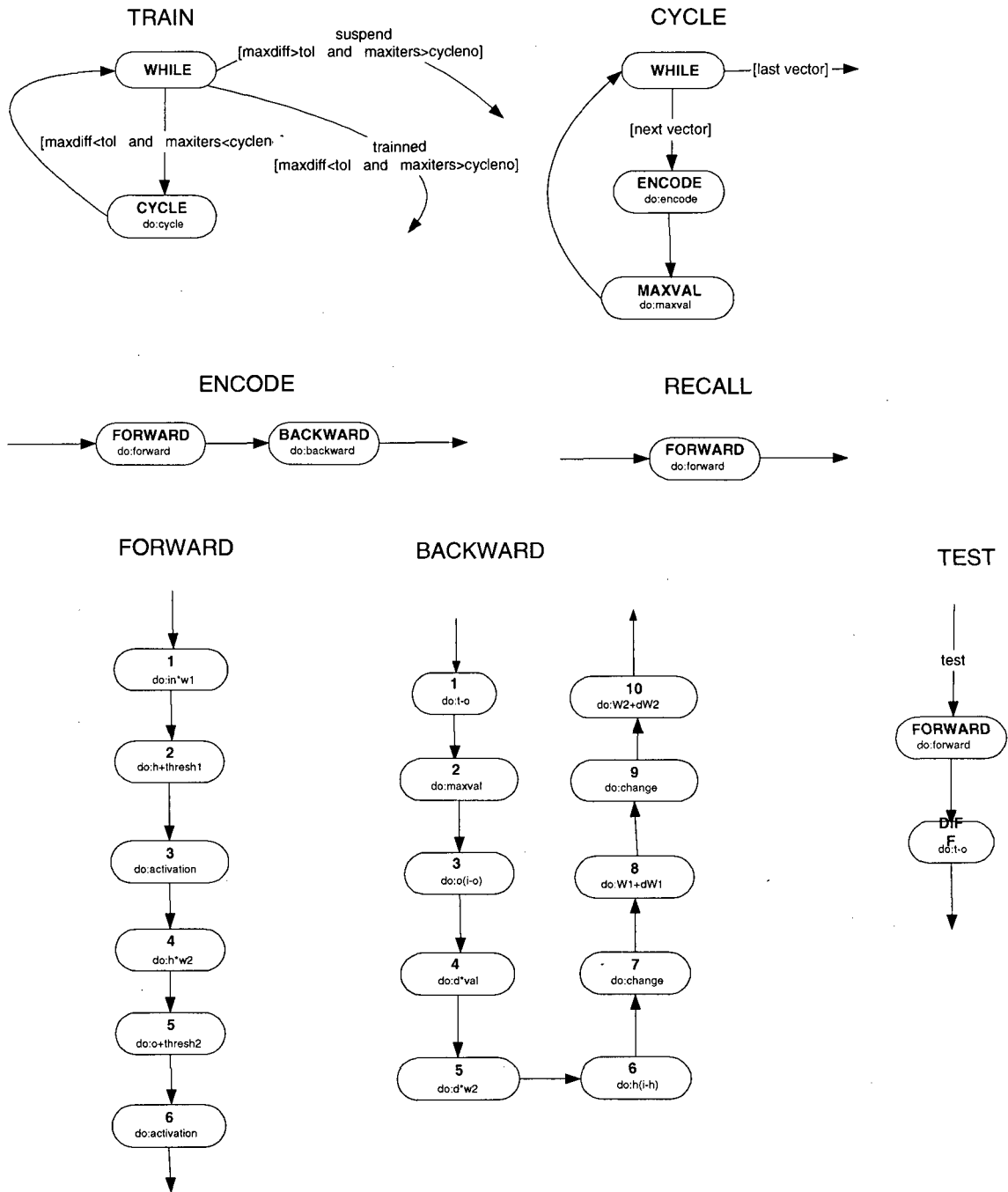


Diagrama de Estados - Neurolib

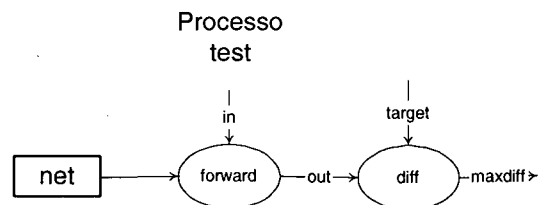
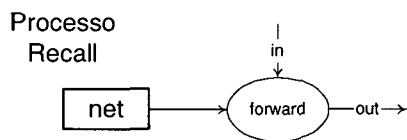
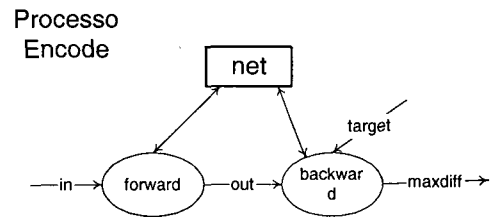
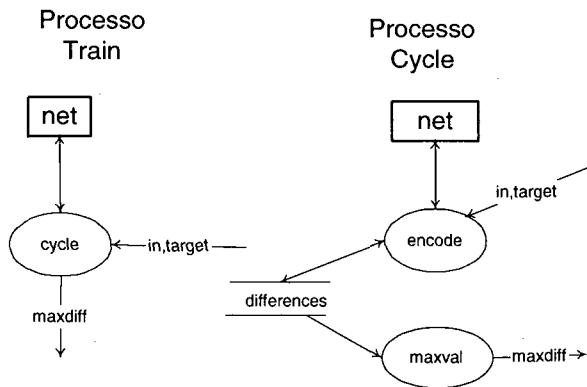
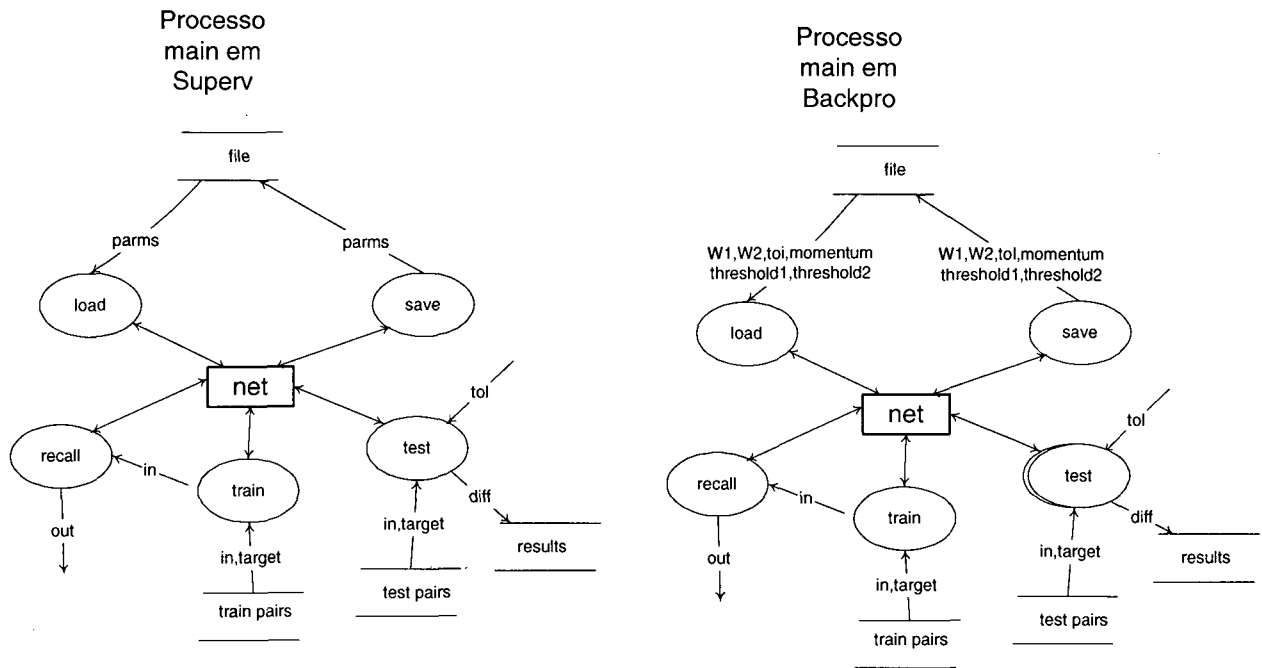
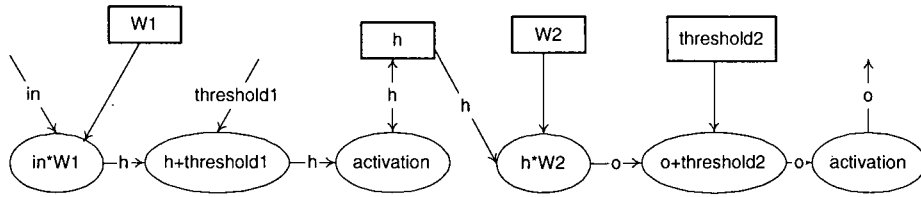
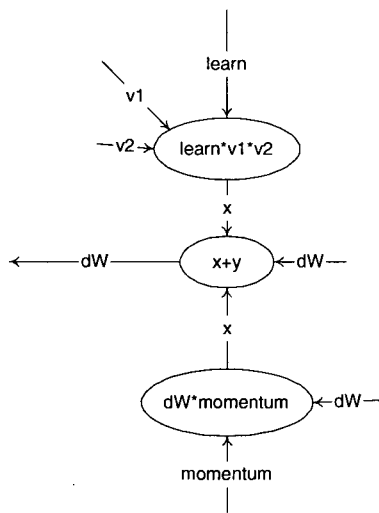


Diagrama deFluxo de Dados - Neurolib

Processo Forward



Processo Change



Processo Backward

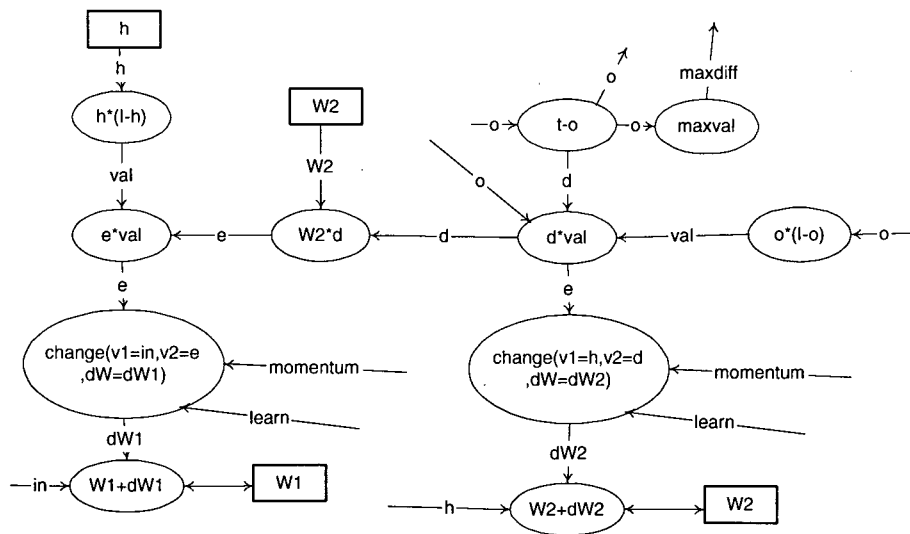


Diagrama deFluxo de Dados - Neurolib

ANEXO C: ARQUIVOS INCLUDES PARA C++ DA SNMPLIB

```

/*****
// Class sBase
// Classe basica para todas as demais da SNMPLib
/*****
#ifndef ClassSBase
#define ClassSBase
#include "../basic/bbase.h"
#include "../file/fmessage.h"
static fMessage sBaseDefaultMessage;
class sBase : virtual public bBase
{
    public:
        sBase()
        {
            bBase::msg = &sBaseDefaultMessage;;
        }
};
#endif

/*****
// Class sListInt
// Implementa o tipo abstrato Lista de Inteiros
/*****
#ifndef ClassSListInt
#define ClassSListInt
#include <ieqseq.h>
#include "sbase.h"
class sListInt : public sBase, public IEqualitySequence<unsigned int>
{
    protected:
        static long sListInt::FCompare(unsigned int const &i, unsigned int const &j);
    public:
        sListInt();
        long compare(sListInt const& list) const;
};
#endif

/*****
// Class sMO
// Classe que modela um Objeto Gerenciavel
/*****
#ifndef ClassSMO

```

```

#define ClassSMO
#include "ste.h"
#include "soid.h"
#include "stree.h"
class sMO : public sBase
{
    public:
        sMO();
        virtual int procGet(sTE& te,const sOID& oid);
        virtual int procSet(sTE& te,const sOID& oid);
        virtual void getListOT(sTree& t);
};
#endif

/*****
// Class sModuleMIB
//  Modulo de Definicoes da MIB privada. Contem os
//  atributos gerenciaveis (instancias de sOT)
*****/
#ifndef ClassSModuleMIB
#define ClassSModuleMIB
#include <imap.h>
#include <iglobals.h>
#include "soid.h"
#include "sot.h"
#include "sos.h"
#include "sbase.h"
#include "stree.h"
#include <istring.hpp>
class sModuleMIB : public sBase
{
    protected:
        sOID root;
        sTree tree;
    public:
        sModuleMIB();
        sOID const & getRoot() const { return root; };
        int load(IString file,sOID& r,sTree t);
        int load(IString file,sTree t);
        IBoolean addOT(sOT * sot);
        IBoolean removeOT(sOT * sot);
        int halt();
        int locateOT(sOT* &sot,sOID const& oid);
        int locateNextOT(sOT* &sot,sOID const& key);
};
#endif

/*****
// Class sOID
//  Identificador de Objetos como definido na SMI da Internet
*****/
#ifndef ClassSOID
#define ClassSOID
#include <iglobals.h>
#include "slistint.h"
#include "defs.h"

```

```

#include "sbase.h"
#include <istring.hpp>
class sOID : public sBase
{
protected :
    sListInt list;
public:
    sOID();
    sOID(const IString &s);
    sOID(const sOID &oid);
    sOID(const OID &oid);
    OID asOID() const;
    sOID& operator=(sOID const& oid);
    IBoolean operator<(sOID const& oid) const;
    IBoolean operator==(sOID const& oid) const;
    sOID operator-(sOID& oid); // prec-cond: a-b => a.length()> b.length()
    sOID operator+(sOID const& oid);
    int length();
    IBoolean shift(int n);
    IString asString() const;
    int tail();
    int head();
    operator OID(); // operador de conversao
    ~sOID();
};
static const sOID NULLSOID;
#endif

/*****
// Class sOS
// Sintaxe de Objeto. A implementacao segue a macro ASN.1
// da SMI Object-Type
*****/
#ifndef ClassSOS
#define ClassSOS
#include "ste.h"
#include "sbase.h"
#include "soid.h"
#include <i0string.hpp>
class sOS : public sBase
{
protected:
    IString name;
public:
    sOS(char *syntaxname=NULL);
    sOS(const sOS &os);
    IString const & getName() const
    {
        return name;
    }
    virtual void copy(sOS *o); // copy the instance to *o

    IBoolean sOS::operator==(sOS const& o) const;
// virtual sOS& operator=(sOS const &o);
virtual int encode(sTE & ste) {return 0;}; // SOS->TE
virtual int decode(sTE & ste) {return 0;}; // TE->SOS
    virtual ~sOS();
};

```

```

};
#endif
//*****
// Class sSOSInteger
// Tipo INTEGER do ASN.1 definido pela SMI
//*****
#ifndef classSOSInteger
#define classSOSInteger
#include "sos.h"
class sOSInteger: public sOS
{
protected:
    int value;
public:
    sOSInteger();
    sOSInteger(int i)
    {
        value = i;
    }
    int encode(sTE & te);
    int decode(sTE & te);
    void copy(sOS *o);
    operator int() const {return value;};
    sOSInteger& operator=(sOSInteger const &o);
    sOSInteger& operator=(int const &i)
    {
        value = i;
        return *this;
    }
};
#endif

//*****
// Class sOT
// Tipo de Objeto. Define um atributo gerenciavel de um objeto
//*****
#ifndef ClassSOT
#define ClassSOT
#include "sos.h"
#include "soid.h"
#include "sbase.h"
#include <iglobals.h>
#include <istring.hpp>
class sOT : public sBase
{
public :
    enum ACCESS {ReadOnly,WriteOnly,ReadWrite,None};
protected:
    IString name;
    ACCESS access;
    sOID oid;
    sOS *os;

public:
// sOT(char *otname=NULL,sOS *o=NULL);
sOT(IString const &OID,IString const &otname,ACCESS acc = ReadOnly,sOS *o=NULL);
sOT(sOT const & sot);

```

```

sOT(IString const& otname);
IString const & getName() const { return name;};
sOT& operator=(sOT const& ot);
IBoolean operator==(sOT const& ot) const;
IBoolean operator<(sOT const& ot) const;
IBoolean allowGetRequest() // check if sender has getRequest permission
{
    return ((access == ReadOnly) || (access == ReadWrite));
}
IBoolean allowGetNextRequest() // check if sender has getNextRequest permission
{
    return ((access == ReadOnly) || (access == ReadWrite));
}
IBoolean allowSetRequest() // check if sender has setRequest permission
{
    return access == ReadWrite;
}
IBoolean grant(Access acc);
sOID const& getOID() const { return oid; };
sOS * getOS() const { return os;};
virtual ~sOT();
};
inline sOID const& key( sOT* const &ot)
{
    return ot->getOID();
}

#endif

/*****
// Class sPDU
// Implementa uma PDU SMUX/SNMP
*****/
#ifndef ClassSPDU
#define ClassSPDU
#include <unistd.h>
#include "defs.h"
#include "sbase.h"
#include "svarbind.h"
#include "svblist.h"
class sPDU : public sBase
{
public:
    enum ERROR_STATUS
    {
        noError = int_SNMP_error__status_noError,
        tooBig = int_SNMP_error__status_tooBig,
        noSuchName = int_SNMP_error__status_noSuchName,
        badValue = int_SNMP_error__status_badValue,
        readOnlyAccess = int_SNMP_error__status_readOnly,
        genErr = int_SNMP_error__status_genErr,
        noToOK
    };
    enum TYPE_MESSAGE
    {
        simple = type_SMUX_PDUs_simple,
        close = type_SMUX_PDUs_close,

```

```

        registerRequest = type_SMUX_PDUs_registerRequest,
        registerResponse = type_SMUX_PDUs_registerResponse,
        getRequest = type_SMUX_PDUs_get__request,
        getNextRequest = type_SMUX_PDUs_get__next__request,
        getResponse = type_SMUX_PDUs_get__response,
        setRequest = type_SMUX_PDUs_set__request,
        trap = type_SMUX_PDUs_trap,
        commitOrRollBack = type_SMUX_PDUs_commitOrRollback
    };
    enum SUBTYPE_MESSAGE { responseFailure,commit,rollBack,responseOK };

protected:
    int request_id;
    ERROR_STATUS error_status;
    TYPE_MESSAGE type_message;
    int error_index;
    sVarBindList variable_bindings;
    SUBTYPE_MESSAGE subtype_message;
public:
    sPDU() {};
    sPDU(int reqId,ERROR_STATUS errorStatus,TYPE_MESSAGE typeMsg,
        int errorIndex,SUBTYPE_MESSAGE subtypeMsg);
    void set(int reqId,ERROR_STATUS errorStatus,TYPE_MESSAGE typeMsg,
        int errorIndex,SUBTYPE_MESSAGE subtypeMsg);
    addVB(sVarBind const &vb);
    void set(type_SNMP_SMUX__PDUs *e);
    int getID() { return request_id; };
    int getErrIndex() { return error_index;};
    TYPE_MESSAGE getType() { return type_message ;};
    ERROR_STATUS getStatus() { return error_status; };
    SUBTYPE_MESSAGE getSType() { return subtype_message; };
    void getVBList(sVarBindList& list ) { list = variable_bindings; };

    void setErrorStatus(ERROR_STATUS errorStatus)
    {
        error_status = errorStatus;
    };
    void setErrorIndex(int errorIndex)
    {
        error_index = errorIndex;
    };
    void setVBList(sVarBindList const& vblist);
    IBoolean isCommitResponse()
    {
        if(subtype_message == commit)
        {
            return True;
        }
        else
        {
            return False;
        }
    };
    IBoolean isResponseFailure()
    {
        if(subtype_message == responseFailure)
        {

```

```

        return True;
    }
    else
    {
        return False;
    }
};

#endif

/*****
// Class sPending
// Implementa um pedido Set pendente
*****/
#ifndef ClassSPending
#define ClassSPending
#include "sos.h"
#include "sot.h"
#include "sbase.h"
#include <iglobals.h>
class sPending : public sBase
{
protected:
    sOT *sot;
    sOS *sos;
public:
    sPending(sOT* ot,sOS * os)
    {
        setLabel("sPending");
        sot = ot;
        sos = os;
    }
    sOT* getOT() { return sot; };
    sOS* getOS() { return sos; };
    sPending& operator=(sPending const& pend);
    IBoolean operator==(sPending const& pend) const;
};

#endif

/*****
// Class sSagent
// Implementa uma subagente SMUX
*****/
#ifndef ClassSSagent
#define ClassSSagent
#include "sbase.h"
#include "soid.h"
#include "smodule.h"
#include "ssmux.h"
#include "smonitor.h"
#include "spending.h"
#include "spdu.h"
#include "stree.h"
#include "sos.h"
#include <istring.hpp>
class sSagent : public sBase
{

```

```

protected :
    sOID identify; // is the unique object-identifer in dotted decimal
    IString name; // is the name of the process acting as an SMUX peer
    IString password; // specifies the password that the snmpd daemon
                    // requires from the SMUX peer client to authenticate
                    // the SMUX association
    IString description;
    sSMUX smux;
    static sModuleMIB moduleMIB;
    IEqualitySequence<sPending> setPendings; // list of Set Operations Pendings

public:
    sSagent(sOID const & o,IString const &n,IString const &passwd,IString const &descr);
    sSagent(sOID id, char *name, char *description);
    int open(char *filename,sOID root,sTree t,int prior=-1,sSMUX::OPERATION
            access=sSMUX::RO);
    int close();
    int dispatch();
    int getMessage(sPDU& const pdu);
    int setMessage(sPDU& const pdu);
    int getNextMessage(sPDU& const pdu);
    virtual int procGet(sOS *os,sOID oid)=0;
    virtual int procSet(sOS *os,sOID oid)=0;
    int commitOrRollBack(sPDU& const pdu);
    int registerResponseTree(sPDU& const pdu);
    virtual ~sSagent();
};
#endif

/*****
// Class sSMUX
// Implementa uma Interface SMUX
*****/
#ifndef ClassSSMUX
#define ClassSSMUX
#include "sbase.h"
#include "spdu.h"
#define TIME_SLICE_WAIT 1
#include "defs.h"
#include "strap.h"
class sSMUX : virtual public sBase
{
private:
    struct type_SMUX_PDUs* event;
    void getPDU(sPDU &pdu);
    void setTE(sTE &te,type_SNMPL_VarBind *v);
    void getTE(sTE &te,type_SNMPL_VarBind *v);

public:
    enum OPERATION {
        del=int_SMUX_operation_delete,RO=int_SMUX_operation_readOnly,RW=int_SMUX_
        operation_readWrite};

protected:
    int fd;

public:
    sSMUX();
    // insere getsmuxEntrybyname:

```



```

        //      Get the smux entry. This entry has to be in /etc/snmpd.conf and
        //      /etc/snmpd.peers (smux....)
        // wait to a smux protocol request
        int wait(sPDU &pdu);// smux_wait
        int init();//smux_init
        int open(sOID& identify,IString &description,IString & passwd);
        int registerTree(sOID tree, int priority,OPERATION operation=RO);
        int close(int reason);
        int trap(sTrap);
        int response(sPDU &spdu);
        int unregisterTree(sOID tree) { return registerTree(tree,-1 ,del); };
};
#endif
#include "sbase.h"
#include <iglobals.h>
#ifdef ClassSTE
#define ClassSTE

//*****
// Class sTE
//  Implementa um Elemento de Transmissão independente da
//  arquitetura da maquina para o ASN.1
//*****
// Class sTE: Type Encode -- Baseado na pagina 168 do SimpleBook
class sTE : public sBase
{
    protected:
        unsigned char tag;
        unsigned char length;
        void * value;
    public:
        sTE::sTE(unsigned char t,unsigned char l,void *v);
        void sTE::set(unsigned char t,unsigned char l,void *v);
        sTE::sTE()
        {
            value = NULL;
            length = 0;
        }
        unsigned char getTag();
        unsigned char getLength();
        void getValue(void *v);
        IBoolean operator==(sTE const& t) const;
        IBoolean isValid();
};
#endif

//*****
// Class sTrap
//  Implementa um Trap SNMP
//*****
#ifdef ClassSTrap
#define ClassSTrap
#include "svblist.h"
class sTrap : public sBase
{
    public:
        enum GENERIC_TRAP

```

```

        {
            coldStart=int_SNMP_generic_trap_coldStart,
            warmStart=int_SNMP_generic_trap_warmStart,
            linkDown=int_SNMP_generic_trap_linkDown,
            linkUp=int_SNMP_generic_trap_linkUp,
            authentFailure=int_SNMP_generic_trap_authenticationFailure,
            egpNeighborLoss=int_SNMP_generic_trap_egpNeighborLoss,
            enterpriseSpecific=int_SNMP_generic_trap_enterpriseSpecific
        };

protected:

    GENERIC_TRAP generic_trap;
    integer specific_trap;
    sVarBindList variable_bindings;

public:
    sTrap(GENERIC_TRAP genericTrap,int specificTrap=0,
          sVarBindList &vblist=NULLVBLIST);
    GENERIC_TRAP const& getGType() { return generic_trap; };
    integer const& getSType() { return specific_trap; };
    sVarBindList const& getVBList() { return variable_bindings; };
};
#endif

/*****
// Class sTree
// Implementa uma arvore cujos nos sao instancias de sOT
/*****
#ifdef classSTree
#define classSTree
#include <isrtmap.h>
#include "sot.h"
#include "soid.h"
class sTree : public ISortedMap<sOT*,sOID>
{
};
#endif

/*****
// Class sVarBind
// Implementa um para (sOID,sTE) usado na PDU SNMP
/*****
#ifdef ClassSVarBind
#define ClassSVarBind
#include <iglobals.h>
#include "ste.h"
#include "sbase.h"
#include "soid.h"
class sVarBind : public sBase
{
protected :
    sOID oid;
    sTE te;

public:
    sVarBind();
    sVarBind(sOID& o, sTE& p);
    sVarBind(const sVarBind &vb);

```

```

        virtual ~sVarBind();
        sOID& getOID() { return oid; };
        sVarBind& operator=(sVarBind const &vb);
        IBoolean operator==(sVarBind const &vb) const;
        void setTE(sTE const &p)
        {
            te = p;
        };
        sTE const& getTE()
        {
            return te;
        };
};
#endif

/*****
// Class sVarBindList
//   Implementa uma Lista de instâncias sVarBind
/*****
#ifndef ClassSVarBindList
#define ClassSVarBindList
#include <ieqseq.h>
#include "svarbind.h"
#include "defs.h"
class sVarBindList : public sBase, public IEqualitySequence<sVarBind>
{
    public:
        sVarBindList()
        {
            setLabel("sVarBindList");
        }
};
static sVarBindList NULLVBLIST;
#endif

```

ANEXO D : ARQUIVOS INCLUDES PARA C ++ DA NEUROLIB

```

// *****
// Classe nBase
// *****
#ifndef ClassNBase
#define ClassNBase
#include "../basic/bbase.h"
#include "../file/fmessage.h"
static fMessage nBaseDefaultMessage;
class nBase : virtual public bBase
{
    public:
        nBase()
        {
            bBase::msg = &nBaseDefaultMessage;
        }
};
#endif

// *****
// Classe nNet
// *****
#ifndef ClassNNet
#define ClassNNet
#include "../applwin/logger.hpp"
#include "vecmat.h"
#include "../applwin/graphic.hpp"
#include "nbase.h"

class nNet : virtual public nBase
{
    protected:
        float tolerance;
        graphic *gph;
        logger *Log;
    public:
        nNet(float tol)
        {
            setLabel("nNet");
            tolerance = tol;
            gph = NULL;
            Log = NULL;
        };
        int associate(graphic* gph1)
        {
            gph = gph1;
            return 0;
        }
        int associate(logger* msg1)

```

```

        {
            Log = msg1;
            return 0;
        }
        virtual int recall() = 0;
};
#endif

// *****
// Classe nSuperv
//     Redes feedforward com aprendizado supervisionado
// *****
#ifndef classNSuperv
#define classNSuperv
#include <iostream.h>
#include "vecmat.h"
#include "nnet.h"

class nSuperv : virtual public nNet
{
protected:
    int n;
    int p;
    float learnrate; //taxa de aprendizagem
    int iters;
    int cycleno;
    virtual int save(char *savefile) = 0;
    virtual int load(char *savefile) = 0;
    virtual int encode(vec &in, vec &out);

public:
    // if return of superv is: 0:train not suspended, 1:train suspended
    nSuperv();
    int set(int nin, int nout, float tol, float learnRate);
    nSuperv(int nin, int nout, float tol, float learnRate);
    float cycle(vec* * &in, vec* * &target, int nfct);
    int train(vec* * &in, vec* * &target, int nfct);
    int getiters(void)
    {
        return iters;
    }
    int getCycle()
    {
        return cycleno;
    }
};
#endif

// *****
// Classe BP,
//     Implementa uma rede neural com algoritmo de aprendizado
//     Backpropagation, ou seja, retro-alimentacao de erros.
// *****
#include "nsuperv.h"
#include "nvpairlist.h"

```

```

#define NUMAXFCT 100
class nBP : virtual public nSuperv
{
    protected:
        int q; // size of hidden layer
        matrix *W1,*W2; // synapse weight matrices
        matrix *dW1,*dW2; // used to compute changes to matrices
        vec *h,*o,*d,*e,*thresh1,*thresh2;
        vec *plot1; // vector to plot in graphic
        vec *plot2; // auxiliar conatins maxx
        vec *plot3; // auxiliar contains mins
        vecpair *minvecs,*maxvecs;
        float momentum,intrange;
        int encode(vecpair &pair); // store one pattern pair
        int encode(vecpair &pair,int index);
        int innum; // interaction number
    public:
        nBP();
        int set(int nin, int nout,int nhidden, float tol,float learnRate,
            int mom, float init,vecpair& minvs,vecpair& maxvs);
        nBP(int nin, int nout,int nhidden, float tol,float learnRate,
            int mom, float init,vecpair& minvs,vecpair& maxvs);
        ~nBP();
        int train(nVecPairList &vplist,unsigned int endcycle=0x7ffffff);
        int test(nVecPairList &vplist);
        int recall(vec &vout,vec& vin); // recall an output pattern given an input
        float cycle(nVecPairList &vplist);
        int save(char *savefile);
        int load(char *savefile);
};

// *****
// Classe vec,matrix e vecpair
// Classes auxiliares a Neurolib. Implementam os tipos de dados vetor, matrix e pares
// de vetores
// *****
#ifdef vecmatClass
#define vecmatClass
#include <iglobals.h>
extern "C"
{
    #include <stdlib.h>
    #include <fcntl.h>
    #include <stdio.h>
    #include <string.h>
    #include <limits.h>
    #include <ctype.h>
    #include <math.h>
    #include <time.h>
    #include <float.h>
    #include <sys/stat.h>
};
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
// C++ doesn t have min/max
#define MAX(a,b) ((a) > (b)) ? (a) : (b)

```

```

#define MIN(a,b) ((a) < (b)) ? (a) : (b)
#include <debug.h>

#include "../basic/bib.h"

double logistic(double activation);
// wil be changed to values much higher than these
const ROWS = 64; //number of rows (length of first pattern)
const COLS = 64; //number of columns (length of second pattern)
const MAXVEC = 64; //default size of vectors
const RAND_PREC = 100; // valor da precisao dos pesos aleatorios - elvis/solange

class matrix;
class vec
{
    friend class bp;
    friend ostream& operator<<(ostream& s,const vec& v1);
    friend class matrix;
    friend class matint;
    friend istream& operator>>(istream& s,const vec& v1);
    int n;
    float *v;

public:
    vec(int size = MAXVEC, int val = 0); //constructor
    int save(FILE *fh);
    int load(FILE *fh);
    ~vec();
    vec(const vec &v1); // copy-initializer
    void set(int i);
    void assign(int index, float value)
    {
        if(index > n)
        {
            cout << "index off range in assign\n";
        }
        else
        {
            v[index] = value;
        }
    }
    matrix matrixXvec();
    int length();
    float distance(vec& A);
    vec& normalize();
    vec& normalizeon();
    vec& randomize(float range);
    vec& scale(vec& minvec, vec& maxvec);
    vec& scaleoff(vec& minvec, vec& maxvec);
    // do product of vector and complement
    float d_logistic();
    float maxval();
    float minval();
    vec& garble(float noise);
    vec& operator=(const vec& v1); // vector assignment
    vec operator+(const vec& v1); // vector addition
    vec operator+(const float d); //vector additve-assignment

```

```

vec& operator+=(const vec& v1);
// supplief for completeness, but we do not use this now
// vector multiply by constant
vec& operator*= (float c);
// vector transpose multiply needs access to v array
IBoolean operator==(const vec& v1) const;
int size() { return n; }; // Incluído ES
float operator[](int x);
int maxindex();
vec& getstr(char *s);
float sumOfElem();
float sumAbsOfElem();
float average()
{
    return sumAbsOfElem()/n;
}
void putstr(char *s);
vec& fourier(vec v); // transformada de fourier
vec operator-(const vec& v1); //subtraction
vec operator-(const float d); //subtraction
float operator*(const vec& v1); //dot-product]
vec operator*(float c); //multiply by constant
vec& sigmoid(vec& v1);
}; // vector class

class vecpair;

class matrix
{
    friend class hop;
    friend class tsp;
    friend ostream& operator<<(ostream& s, matrix& m1);
    friend istream& operator>>(istream& s, matrix& m1);
protected:
    float **m; // the matrix representation
    int r, c; // numberof rows and columns
public:
    // constructors
    matrix(int n=ROWS, int p=COLS, float range=0);
    matrix(int n, int p, float value, float range);
    matrix(int n,int p, char *fn);
    matrix(const vecpair& vp);
    matrix(matrix& m1);
    ~matrix();
    float get(int i,int j);
    int depth();
    int width();
    matrix& operator=(const matrix& m1);
    matrix operator+(const matrix& m1);
    matrix operator-(const matrix& m1);
    int range(const float min,const float max);
    matrix operator*(const matrix& m1);
    double sumOfElem();
    vec operator*(vec& v1);
    vec colslice(int col);
    vec rowslice(int row);
    void insertcol(vec& v, int col);

```



```

void insertrow(vec& v, int row);
int closestcol(vec& v);
int closestrow(vec& v);
int closestrow(vec& v, int *wins, float scaling);
int load(FILE *fh);
int save(FILE *fh);
vec vecXmatrix();
matrix& operator+=(const matrix& m1);
matrix& operator-(const float d);
matrix& operator*(const float d);
matrix& operator/(const float d);
matrix& operator*=(const float d);
void initvals(const vec& v1,const vec& v2,
const float rate = 1.0, const float momentum =0.0);
int delta1(matrix& x,float inib);
int delta(matrix& x,float inib,int boundary);
void fourier();
void assign(int i,int j,float d);
void conv(float alfa);
void transfer();
void readint(istream& s);
};

class vecpair
{
protected:
    friend class matrix;
    friend istream& operator>>(istream& s, vecpair& v1);
    friend ostream& operator<<(ostream& s,vecpair& v1);
    friend matrix::matrix(const vecpair& vp);
        // flag signalling whether encoding succeeded
    int flag;
public:
    vec *a;
    vec *b;
    vecpair(int n = ROWS, int p = COLS);
    vecpair(int n,int p,int val);
    vecpair(vec& A, vec& B);
    vecpair(const vecpair& AB);
    ~vecpair();
    vec getA()
    {
        return *a;
    }
    vec getB()
    {
        return *b;
    }

    vec *getApA()
    {
        return a;
    }
    vec *vecpair::getApB()
    {
        return b;
    }
};

```

```

    }
    vecpair& operator=(const vecpair& v1);
    IBoolean operator==(const vecpair &v1) const;
    int save(FILE *fh);
    int load(FILE *fh);
    vecpair& scale(vecpair& minvecs, vecpair& maxvecs);
    vecpair& scaleoff(vecpair& minvec, vecpair& maxvec);
};
#endif

// *****
// Classe nVecList
// implementação do tipo abstrato lista de vetores
// *****

#ifndef ClassNVecList
#define ClassNVecList
#include <ieqseq.h> // Importada da Biblioteca IBM Class Library
#include "../math/vecmat.h"
class nVecList : public IEqualitySequence<vec>
{
protected:
    int veclen; // Tamanho dos vetores ( deve ser fixo )
public:
    nVecList(int vlen)
    {
        veclen = vlen;
    }
    friend istream & operator>>(istream& i, nVecList &v1);
    friend ostream & operator<<(ostream& o, const nVecList &v1);
};
#endif

#ifndef ClassNPairVecList
#define ClassNPairVecList
#include <ieqseq.h> // Importada da Biblioteca IBM Class Library
#include "../math/vecmat.h"
class nVecPairList : public IEqualitySequence<vecpair>
{
protected:
    int veclen1; // Tamanho dos vetores ( deve ser fixo )
    int veclen2;
public:
    nVecPairList(int vlen1,int vlen2)
    {
        veclen1 = vlen1;
        veclen2 = vlen2;
    }
    friend istream & operator>>(istream& i, nVecPairList &v1);
    friend ostream & operator<<(ostream& o, const nVecPairList &v1);
    nVecPairList& scale(vecpair& minvecs, vecpair& maxvecs);
};

```

BIBLIOGRAFIA

- 01 ALEKSANDER, I. e MORTON, H. An Introduction to Neural Networking. London: Chapman & Hall, 1990.
- 02 BOOCH, G. Object-Oriented Software Construction. Hertfordshire: Prentice Hall, 1988.
- 03 BACH, M. J. The Design of The UNIX Operating System. New Jersey: Prentice Hall, 1993.
- 04 BLUM, A. Neural Networks In C++ - An Object-Oriented Framework for Building Connectionist Systems. New York: John Wiley & Cons, 1992.
- 05 COX, B. J. Object-Oriented Programming. Reading: Addison-Wesley, 1986.
- 06 COMER, D.E e STEVENS, D. L. Internetworking with TCP/IP, v.1. New Jersey: Prentice Hall.
- 07 COMER, D.E e STEVENS, D. L. Internetworking with TCP/IP, v.2. New Jersey: Prentice Hall.
- 08 CONGRESSO BRASILEIRO DE REDES NEURAIIS. (1.:1994: Itajubá). Anais. Itajubá:CEMIG, 1994.
- 09 GROSSBERG, S. Adaptative Pattern Recognition and Universal Recoding: Part I, Parallel Development and Coding of Neural Feature Detectors. Biol. Cybern., 1976.
- 10 HEBB, D. O. The Organization of Behaviour, New York: Wiley, 1949.
- 11 KERNIGHAN, B. W. e PIKE, R. The UNIX Programming Environment. Englewood Cliffs: Prentice-Hall, 1984.
- 12 KOHONEN, T. Self-Organization and Associative Memory. Heidelberg: Springer-Verlag, 1988.
- 13 KOSKO, B. Neural Networks and Fuzzy Systems - A Dynamical Systems Approach to Machine Intelligence. New Jersey: Prentice-Hall Inc., 1992.
- 14 LIEBERHERR, K. e HOLLAND e RIEL, A. Object-Oriented Programming: na Objective Sense of Style. OOPSLA '88 as ACM SIGPLAN 23, 1988.
- 15 PAO, Y-H. Adaptive Pattern Recognition and Neural Networks. New York: Addison Wesley Publishing Company, 1989.
- 16 MEYER, B. Object-Oriented Software Construction. Prentice Hall.

- 17 MINSKY, M. e PAPERT, S. Perceptrons: na Introduction to Computational Geometry. Massachusetts: MIT Press, 1969.
- 18 ROSE, M. T. The Open Book - A Pratical Perspective on OSI. New Jersey: Prentice Hall, 1991.
- 19 ROSE, M. T. The Simple Book - An Introdution to Management of TCP/IP - based internets. New Jersey: Prentice Hall, 1991.
- 20 ROSE, M. T. The Simple Book - An Introdution to Internet Management- Second Edition. New Jersey: Prentice Hall, 1991.
- 21 ROSE, M. T. Structure and Identification of Management Information for TCP/IP based internets. Request for Comments 1156. DDN, 1990.
- 22 ROSE, M. T. Management Information Base Network of TCP/IP based internets: MIB-II. Request for Comments 1158. DDN, 1990.
- 23 ROSE, M. T. SNMP MUX Protocol and MIB. Request for Comments 1227. DDN, 1991.
- 24 RUMBAUGH, J. et al. Object-Oriented Modeling And Design. New Jersey: Prentice Hall, 1991.
- 25 RUMELHART, D. ét al. Learning Representations by Backpropagation Errors. Nature, 1986.
- 26 RUMELHART, D. e McCLELLAND, James L. Paralel Distributed Processing, vols. I e II. Cambridge: MIT Press, 1989.
- 27 SCHOFFSTALL, M. L. e CASE, J. D e FEDOR, M. S A Simple Network Management Protocol. Request for Comments 1157. DDN, 1990.
- 28 SARI, S.T. Protótipo de Um Sistema de Reconhecimento de Padrões Conexionista Híbrido. Florianópolis, 1994. Dissertação de Mestrado em Engenharia de Produção - Universidade Federal de Santa Catarina.
- 29 SARI, S.T. e Loesh, C. Redes Neurais Artificiais - Fundamentos e Modelos. Blumenau: Editora da FURB, 1996.
- 30 SIMPÓSIO BRASILEIRO DE REDES DE COMPUTADORES. (13.:1995:Belo Horizonte) Anais. Belo Horizonte: UFMG.
- 31 SIMON, H. Neural Networks - A Comprehensive Foundation. Hamilton: Macmillan College Publishing Company, 1994.
- 32 SIMPSON, P. K. Artificial Neural Systems. Elmsford: Pergamon Press, 1990.

- 33 SIMPSON, P. K. Associative Memory Systems. Proceedings of the International Joint Conference on Neural Networks, 1990.
- 34 SOARES, L.F e LEMOS, G. e COLCHER S. Redes de Computadores - Das LANS, MANS e WANS às Redes ATM. Rio de Janeiro: Editora Campus, 1995.
- 35 STALLINGS, W. Data Communications. Second Edition. Macmillan Publishing Company, 1985.
- 36 STALLINGS, W. Local and Metropolitan Area Networks. Fourth Edition. Macmillan Publishing Company, 1993.
- 37 STEVENS, W. R. Advanced Programming in the UNIX Environment. Addison Wesley, 1993.
- 38 STROUSTRUP, B. e ELLIS, M. C++ - Manual de Referência Comentado. Rio de Janeiro: Editora Campus, 1993.
- 39 TANENBAUM, A. S. Redes de Computadores - Segunda Edição. Rio de Janeiro: Editora Campus, 1994.
- 40 TOWNSEND, R. L. SNMP Application Developer's Guide. New York: International Thomsom Publishing , 1995.
- 41 VIEIRA, E.M. e MOSCHETTA, M.C. e WESTPHALL, C. B. Controle de Acesso em Gerência de Segurança para a redeUFSC - Anais do SBRC 95. Belo Horizonte.
- 42 WASSERMAN, P.D. Neural Computing - Teory and Praticce. New York: Van Nostrand Reinhold, 1989.
- 43 WIDROW, B. 30 Years of Adaptive Neural networks: Perceptron, Madaline, and Backpropagation. IEEE, Vol. 78, n. 9, 1990. P.1415-42.
- 44 YOURDON, E. Modern Structured Analisys. Englowood Cliffs: Yourdon Press, 1989.

GLOSSÁRIO

- ASN.1** - *Abstract Syntax Notation One* - A linguagem da OSI para sintaxe de descrição abstrata.
- BAM** - *Bidirectional Associative Memory* - É uma arquitetura de rede neural heteroassociativa capaz de generalização, produzindo saídas corretas, apesar das entradas estarem parcialmente corrompidas.
- Delta** - Regra Delta Generalizada - Também conhecida como Algoritmo de Correção de Erros *Backpropagation*, prove uma solução para o problema de associação de padrões não-lineares em uma rede neural, substituindo as funções de ativação por funções contínuas sigmoidais.
- Híbridas** - As redes neurais híbridas criam uma macro estrutura completa que consistem em múltiplas redes neurais interagindo, cada uma realizando uma tarefa vital e única na resolução de um problema complexo.
- Hopfield** - Rede neural auto-associativa, desenvolvida em 1986, por Hopfield.
- IEEE** - *Institute of Electrical and Electronics Engineers* - Uma organização profissional que realiza, como parte de suas funções, padronizações para a OSI.
- IAB** - *Internet Architecture Board* - O corpo técnico que supervisiona o desenvolvimento do conjunto de protocolos da Internet.
- Internet** - Uma rede composta por outras (sub)redes. Atualmente, é a maior rede existente.
- IP** - *Internet Protocol* - O protocolo de rede usado na Internet.
- ISO** - *International Organization for Standardization* - Uma organização que produz muitos dos padrões adotados mundialmente.
- ISODE** - *ISO Development Environment* - Uma ferramenta de pesquisa desenvolvida para estudar as camadas superiores do modelo OSI.
- IETF** - *Internet Engineering Task Force* - Uma força tarefa da IAB preocupada com necessidades de curto prazo da Internet.

LAN - *Local Area Network* - Termo usado para denotar uma rede local de computadores.

Kohonen - Mapas de Kohonen - Mapas que preservam a topologia de auto-organização estudados por Teuvo Kohonen em sua obra *Self-Organization and Associative Memory*, 1989.

MIB - *Management Information Base* - Uma coleção de objetos que podem ser acessados via protocolo de gerência de rede.

MODEM - MODulador/DEModulador - Equipamento que transforma sinais digitais em sinais analógicos para serem enviados em circuitos telefônicos.

OMT - *Object Modeling Technique* - Metodologia de Modelagem Orientada a Objetos usada no desenvolvimento de sistemas computacionais.

PE - *Presentation Element* - Uma estrutura de dados capaz de representar um objeto ASN.1.

PDU - Protocol Data Unit - Um objeto de dados trocados pelas máquinas de protocolos, usualmente contendo uma informação de controle de protocolo e dados de usuário.

RFC - *Request For Comments* - A série de documentos descrevendo o conjunto de protocolos da Internet.

SMUX - *SNMP Mux Protocol* . Protocolo de comunicação entre um agente SNMP e um *daemon* que controla objetos gerenciáveis.

SNMP- *Simple Network Management Protocol* - Protocolo de gerenciamento do modelo de gerência Internet.

TCP - *Transmission Control Protocol* - O protocolo de transporte que oferece um serviço orientado a conexão.

UDP - *User Datagram Protocol* - O protocolo de transporte que oferece um serviço não orientado a conexão.