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

UM NÚCLEO MULTITHREADED PARA AGENTES DE GERENCIAMENTO OSI/ISSO

DISSERTAÇÃO SUBMETIDA À UNIVERSIDADE FEDERAL DE SANTA CATARINA PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIA DA COMPUTAÇÃO

Rivalino Matias Júnior

Florianópolis, Junho de 1997

Um Núcleo Multithreaded para Agentes de Gerenciamento OSI/ISO

RIVALINO MATIAS JÚNIOR

Esta Dissertação foi julgada para a obtenção do título de

MESTRE EM CIÊNCIA DA COMPUTAÇÃO

e aprovada em sua forma final pelo programa de Pós-Graduação em Ciência da Computação

Prof. Paulo José de Freitas Filho, Dr.
Orientador

Prof. Elizabeth Sueli Specialski
Co-orientadora

Prof. Murilo Silva de Camargo, Dr.
Coordenador do Curso

Banca Examinadora:

Prof. Paulo José de Freitas Filho, Dr. (Presidente)

Prof. Elizabeth Sueli Specialski, M.Sc.

Prof. Tereza Cristina Melo de Brito Carvalho, Dr.

Prof. Luis Fernando Friedrich, Dr.

Prof. Vitório Bruno Mazzola, Dr.

RESUMO

Nos últimos anos, os requisitos de gerenciamento para redes de computadores tem exigido que plataformas de gerenciamento sejam construídas levando-se em conta os atuais avanços tecnológicos, em função da grande variedade e capacidade dos elementos disponíveis nestes ambientes.

Neste trabalho, é proposto um núcleo *multithreaded* para agentes de uma plataforma de gerenciamento de redes, a qual está sendo implementada seguindo o modelo de gerenciamento OSI/ISO. Conceitos relacionados com gerenciamento OSI/ISO e *multithreading*, juntamente com a implementação de objetos ativos, serão abordados.

Palavras Chave

Gerenciamento de Redes, Plataformas de Gerenciamento de Redes, Desenvolvimento de Agentes de Gerenciamento, Núcleo de Agentes, Objetos Ativos, Multithreading,

ABSTRACT

In recent years, the requirements of computer networks management are demanding that management platforms must be built based on the current technological advances, due to the great variety and capability of the available elements in these environments.

In this work is proposed a multithreaded core to implement management agents in a network management platform, which is being implemented following the OSI/ISO management model. This work will approach the concept related to OSI/ISO management and multithreading, joined with the implementation of active objects.

Keywords

Network Management, Network Management Platforms, Network Management Agent Development, Agent Core, Active Objects, Multithreading,

Agradecimentos

A Deus.

Aos meus pais pelo apoio e incentivo durante toda a minha vida.

Aos orientadores Professor Paulo José de Freitas Filho e Professora Elizabeth Sueli Specialski pelo incentivo e dedicação ao longo de todo período de desenvolvimento deste trabalho.

Aos colegas, professores e funcionários da UFSC que, direta ou indiretamente, contribuíram para a realização deste trabalho.

Ao CNPq pelo apoio financeiro concedido através de uma bolsa de estudos.

Ao meu grande amigo Carlos Antunes pela amizade e incentivo.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

ABREVIATURAS

ARR	- Attributes for Representing Relationships
ACSE	- Association Control Service Element
API	- Application Program Interface
ASE	- Application Service Element
ASN.1	- Abstract Syntax Notation.1
CMIP	- Common Management Information Protocol
CMIPDU	- Common Management Information Protocol Data Unit
CMIS	- Common Management Information Service
CMISE	- Common Management Information Service Element
CMOT	- CMIP Over TCP/IP
FCFS	- First Come First Served
FIFO	- First In First Out
FTAM	- File Transfer, Access and Management
GDMO	- Guidelines for the Definition of Managed Objects
GUI	- Graphical User Interface
IEC	- International Electrotechnical Commission
I/O	- Input/Output
IP	- Internet Protocol
IPC	- Interprocess Communication
ISO	- International Organization for Standardization
ITU	- International Telecommunication Union
LAN	- Local Area Networks
LPP	- Lightweight Presentation Protocol
LWP	- Lightweight Process
MAN	- Metropolitan Area Networks
MIB	- Management Information Base
MIS	- Management Information Service
MIT	- Management Information Tree
MO	- Managed Object
MUTEX	- Mutual Exclusion
OM	- Object Manager
OOP	- Object-Oriented Programming
OSI	- Open Systems Interconnection

OSIE - Open Systems Interconnection Environment
PDU - Protocol Data Unit
POSIX - Portable Operating System Interface
Pthreads - POSIX Threads
PVM - Parallel Virtual Machine
RAM - Random Access Memory
RM-OSI - Reference Model - Open Systems Interconnection
ROSE - Remote Operations Service Element
RPC - Remote Procedure Call
RR - Real Resource
SACF - Single Association Control Function
SARF - Security Alarm Reporting Function
SGBD - Sistema de Gerenciamento de Banco de Dados
SVR4 - System V Release 4
SMAE - System Management Application Entity
SMASE - System Management Application Service Element
SMFA - Specific Management Functional Area
SMI - Structure Management Information
SNMP - Simple Network Management Protocol
SQL - Structured Query Language
TCP - Transmission Control Protocol
TMF - Test Management Function
TMN - Telecommunication Network Management
TS - Thread Safe
UDP - User Datagram Protocol
WAN - Wide Area Network
WWW - World Wide Web
X.500 - OSI Directory Service

CAPÍTULO I

Introdução

1.1 - Apresentação

O modelo de referência para interconexão de sistemas abertos RM-OSI [1] provê uma descrição das atividades necessárias para interconectar sistemas usando meios de comunicação [2]. A interconexão destes vários sistemas abertos cria um ambiente distribuído, onde a quantidade e a variedade de recursos exigem capacidades de gerenciamento que possam manter a qualidade dos serviços adequada aos requisitos de seus usuários.

Para suprir estes requisitos de gerenciamento, a ISO e a ITU-T, ambas organizações que trabalham com padronizações em sistemas abertos, desenvolveram padrões e recomendações internacionais para a área de gerenciamento de sistemas.

O escopo deste trabalho é um ambiente para desenvolvimento de aplicações de gerenciamento, o qual segue o modelo definido nos documentos de padronização internacional para gerenciamento de sistemas abertos.

1.2 - Objetivo

Este trabalho visa definir um núcleo *multithreaded* para a implementação de agentes de gerenciamento OSI. Este núcleo oferece uma implementação *multithreaded* das funções básicas dos processos agentes, ficando a cargo do usuário (projetista da aplicação), somente implementar os comportamentos dos objetos gerenciados. A proposta e implementação do núcleo *multithreaded* faz parte de um projeto que tem como objetivo a definição e implementação de uma plataforma para gerenciamento de redes, a qual segue o modelo de gerenciamento OSI/ISO.

A definição e implementação do núcleo *multithreaded* cria um modelo genérico para a organização das aplicações agentes na plataforma, servindo como base para a implementação das várias funções de gerenciamento e suporte à implementação dos objetos gerenciados.

Em função de atualmente, não ter sido definida nenhuma forma de estruturação para as aplicações agentes, na plataforma, foi necessário a definição de um modelo para sua implementação.

1.3 - Estrutura do Trabalho

Este trabalho está organizado em 7 capítulos, que versam sobre conceitos do modelo de gerência OSI, projeto de uma plataforma para gerência de redes, processos de aplicações de gerenciamento, *multithreading*, e finalmente o modelo de núcleo *multithreaded* e conclusões a respeito dos resultados obtidos.

O capítulo 2 apresenta uma visão geral do Modelo de Gerência OSI, seus componentes, suas funcionalidades, seu modelo de informação e aspecto organizacional.

O capítulo 3 apresenta o projeto que está sendo desenvolvido para uma plataforma de gerência de redes. Cada um dos módulos que compõem o projeto são sucintamente apresentados, descrevendo o estado atual da plataforma e futuros trabalhos relacionados.

O capítulo 4 aborda os processos de aplicação de gerenciamento, definindo suas principais características e necessidades. Processos agentes são abordados em detalhes, visto serem o escopo deste trabalho.

O capítulo 5 apresenta conceitos relacionados com *threads*, visto ser a base para a implementação do núcleo proposto.

No capítulo 6, é apresentada a proposta para um modelo de núcleo de agentes, juntamente com detalhes de sua implementação. Uma comparação com outros modelos de núcleo é apresentada.

No capítulo 7, finalmente são apresentadas as conclusões, dificuldades encontradas no decorrer deste trabalho, e perspectivas para futuros projetos.

CAPÍTULO II

Gerenciamento OSI

2.1- Introdução

Gerenciamento OSI é aplicável a uma larga faixa de ambientes de comunicação e processamento distribuído. Estes ambientes, vão desde interconexão de pequenos sistemas de redes locais (LANs), até a interconexão de redes corporativas de grande escala (MANs e WANs).

Ambientes pequenos podem utilizar características de gerenciamento apropriadas, com somente um gerente capaz de monitorar e coordenar o ambiente gerenciado através de alguns processos agentes. Múltiplos gerentes suportam ambientes de larga escala [2].

Este capítulo aborda, de uma maneira geral, o modelo de gerência OSI, a fim de apresentar conceitos básicos que serão utilizados nos próximos capítulos.

2.2- Modelo de Gerência OSI/ISO

O modelo de gerenciamento para sistemas abertos provê mecanismos para a monitoração, controle e coordenação de recursos dentro de um ambiente OSI (OSI*Environment*) [3].

Estes recursos são vistos, para o propósito de gerenciamento, como objetos gerenciados. Portanto, objetos gerenciados são abstrações de recursos reais do sistema, sejam eles lógicos ou físicos.

Os objetos gerenciados interagindo com seus recursos permitem que entidades de aplicação de gerenciamento de sistemas (SMAE) exerçam controle e monitoração sobre eles. A Figura 2.1 ilustra este conceito.

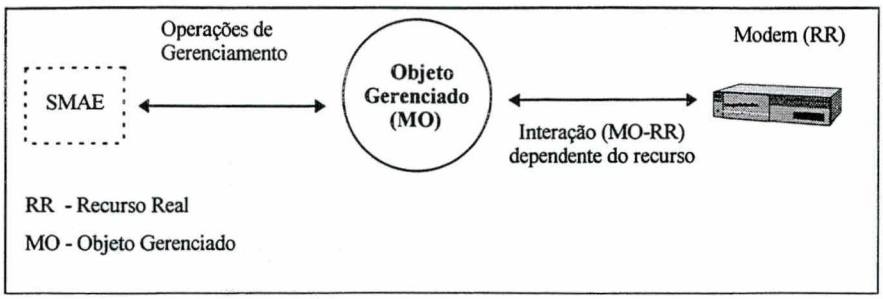

Figura 2.1 - Interações do Objeto Gerenciado.

A fim de gerenciar um ambiente de comunicação (ambiente distribuído), as entidades de aplicação de gerenciamento de sistemas são, por sua vez, distribuídas entre os vários sistemas abertos. A Figura 2.2 apresenta de maneira conceitual, a interação entre os componentes do modelo.

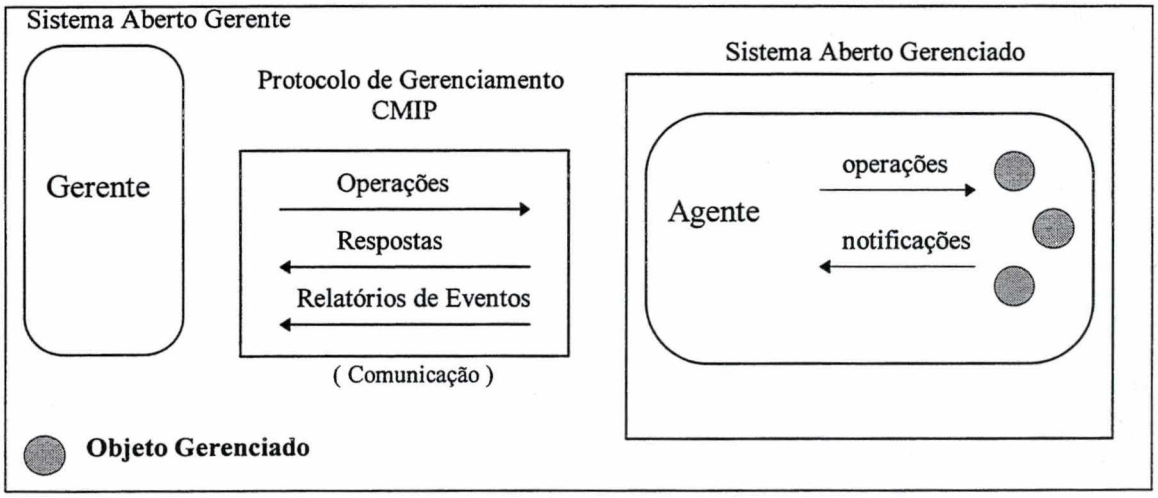

Figura 2.2 - Interações do Gerenciamento de Sistemas.

Como ilustrado na Figura 2.2, as interações entre entidades de aplicação de gerenciamento são abstraídas em termos de operações de gerenciamento (CREATE, DELETE, GET, CANCEL-GET, SET, ACTION) e relatórios de eventos (EVENT-REPORT), sendo esta comunicação efetuada pelo protocolo de gerenciamento CMIP [4].

Aplicações que fazem uso dos serviços, de gerenciamento de sistemas, também chamadas MIS-Users, podem assumir dois papeis:

- Agente;
- Gerente.

Aplicações de gerência no papel de agente têm como objetivo monitorar e controlar os objetos gerenciados dentro de seu ambiente de gerenciamento, executando operações de gerenciamento submetidas pelo gerente e emitindo relatórios de eventos em função das notificações geradas pelos objetos gerenciados.

Aplicações no papel de gerente são a parte da aplicação distribuída responsável por uma ou mais atividades de gerenciamento. Por atividade de gerenciamento entende-se a emissão de operações de gerenciamento e/ou recepção de relatórios de eventos, juntamente com os seus respectivos processamentos.

O conceito de gerente não está limitado às aplicações dedicadas ao gerenciamento de sistemas. Qualquer aplicação que necessite acessar informações relativas ao gerenciamento pode usar os serviços do protocolo de gerenciamento. Ressaltando, no entanto, que estas devem possuir os requisitos mínimos para estabelecerem uma associação de gerenciamento com outros MIS-Users [4].

O modelo de gerência OSI é organizado seguindo quatro aspectos:

- Informação de gerenciamento;
- Protocolo de comunicação;
- Áreas funcionais;
- Aspectos organizacionais.

2.3 - Informação de Gerenciamento

O objetivo do modelo de informação é fornecer uma estrutura para a informação que é transportada pelo protocolo de gerenciamento, e modelar os aspectos de gerenciamento dos recursos (atributos, notificações, comportamento, etc.) que estão sendo gerenciados. Para isto, o modelo de informação trabalha com o conceito de objeto gerenciado.

Objeto gerenciado no modelo de gerência OSI é equivalente ao conceito de instância de uma classe no paradigma orientado a objeto, o qual é adotado para a especificação da informação de gerenciamento. A utilização do paradigma orientado a objeto para a especificação da informação de gerenciamento não necessariamente implica em utilizá-lo na implementação do sistema.

Para proporcionar uma maior modularização da informação de gerenciamento, uma definição de classe de objeto gerenciado é uma coleção de pacotes (*packages*), sendo que cada pacote é uma coleção de atributos, operações, notificações e comportamentos relacionados. Dois tipos de pacotes são definidos:

- Obrigatórios ;
- Condicionais.

Pacotes obrigatórios sempre são instanciados no momento da criação do objeto gerenciado. Pacotes condicionais, como o próprio nome indica, somente serão parte de um objeto gerenciado se uma determinada condição for satisfeita. Tal condição é especificada na definição da classe do objeto (Figura 2.3), a qual está sendo instanciada.

Para fins de documentação das especificações de classes de objetos gerenciados, juntamente com pacotes e características associadas (atributos, notificações, etc.), um conjunto de *templates* são utilizados. *Templates* são módulos de definição que possibilitam ao usuário criar "modelos" (classes de objetos, tipos de atributos, etc.) e posteriormente utilizá-los para geração de várias instâncias (exemplares) destes modelos. Um *template*, que define uma classe de objeto gerenciado, pode ser usado para a criação de vários objetos (instâncias) desta classe.

Existem regras específicas [5] que devem ser respeitadas a fim de se realizar a definição de cada *template*, sendo que o mapeamento destas definições para suas respectivas implementações em linguagens de programação (C, C++, ADA, etc.) pode ser feito de maneira automatizada. Um exemplo de definição de classe de objeto gerenciado está apresentado na Figura 2.3.

```
modemClass  MANAGED OBJECT CLASS                    "Nome da classe de objeto gerenciado"
        DERIVED FROM: top;
        CHARACTERIZED BY      modemPackage;         "Nome do pacote obrigatório"
        CONDITIONAL PACKAGES
          faxModemPackage     PACKAGE               "Nome do pacote condicional"
            ACTIONS           resetFaxModem;        "Nome da ação definida no pacote condicional"
          REGISTERED AS {joint-iso-ccitt ms(9) smi(3) part(4) package(4)
                         ModemPack(0) };            "Registro do pacote condicional"
        PRESENT IF  ! A placa modem suportar funcionalidades de fax !;  "Condição para instanciação do
                                                                         pacote condicional"

REGISTERED AS { joint-iso-ccitt ms(9) smi(3) part4(4) managedObjectClass(3) exemploDeClasse(0) };
"Registro da classe de objeto gerenciado"
```

Figura 2.3 - Exemplo de Definição de Classe de Objeto Gerenciado.

A organização das classes de objetos gerenciados e suas instâncias seguem dois conceitos básicos do modelo de informação:

- Hierarquia de Herança;
- Hierarquia de *Containment*.

Toda classe de objeto gerenciado é derivada direta ou indiretamente da classe *top* [6], formando uma árvore de herança. A classe *top* não é instanciável, esta reúne características que devem estar presentes em todas as subclasses do sistema. Em C++, esta classe pode ser comparada com uma classe base virtual [7].

Uma classe de objeto gerenciado pode ser derivada de uma ou mais classes base, permitido assim o conceito de herança múltipla. Para maiores informações sobre herança ver [5] [6].

Objetos gerenciados de uma classe podem conter outros objetos gerenciados da mesma classe ou de classes diferentes, este relacionamento é chamado de *containment* [6]. O relacionamento de *containment* pode ser usado para representar relacionamentos de objetos do mundo real (Ex. diretório[ALUNOS]<-arquivo[CPGCC]<-registro[#3]). O relacionamento real dos objetos no ambiente gerenciado, não necessariamente deve ser igual à hierarquia de *containment.*

Em todo sistema gerenciado, instâncias das classes de objetos gerenciados são organizadas em uma árvore de informação de gerenciamento (MIT) [6] de acordo com os relacionamentos de *containment,* para propósitos de nomeação destes objetos. Objetos contidos em outros objetos são chamados subordinados, sendo os objetos que

contém chamados objetos superiores. O nome de um objeto é a combinação do nome de seu objeto superior e uma informação que identifica, unicamente, este objeto no escopo de seu objeto superior. Este nome que identifica unicamente o objeto no sistema gerenciado é chamado DN (*Distinguished Name*). Para maiores informações sobre o esquema de nomeação (*naming schema*) ver [6] [8].

No topo da árvore de nomeação ou *containment,* tem-se um objeto chamado *root*, o qual não possui propriedades e sempre existe para fins de referência inicial. As Figuras 2.4 e 2.5 apresentam exemplos de hierarquia de herança e *containment* respectivamente.

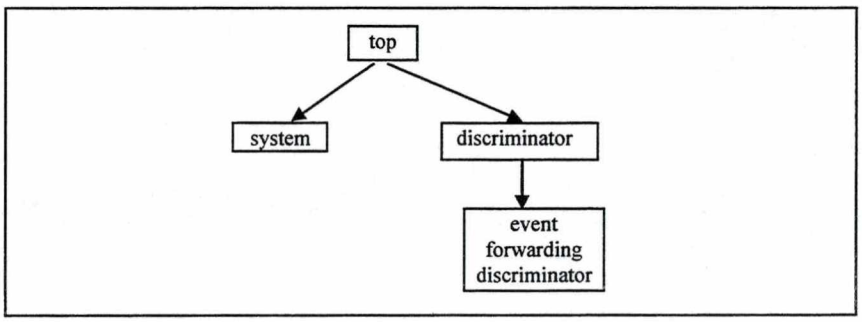

Figura 2.4 - Hierarquia de Herança.

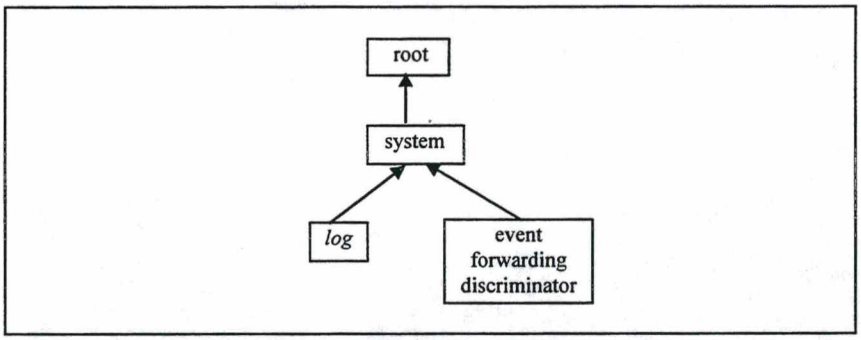

Figura 2.5 - Hierarquia de Containment.

2.4 - Protocolo de Gerenciamento

A interação entre gerente e agente é realizada através de um paradigma cliente-servidor. Aplicações gerentes são clientes de aplicações agentes, as quais exercem o papel de servidores de objetos gerenciados, ressaltando que agentes (servidores) podem emitir relatórios aos seus clientes em função da ocorrência de

algum evento relacionado com seus objetos gerenciados. Em [9], agentes são chamados de servidores de informações gerenciais (MIS[1] - *Management Information Server*).

O Serviço de comunicação definido para gerenciamento de sistemas é o CMIS [10], contudo, MIS-Users podem utilizar outros protocolos de transferência como por exemplo o protocolo FTAM [11][12]. O uso do CMISE implica na utilização do ROSE, pois o CMIP, protocolo que implementa o CMIS, utiliza os serviços do ROSE a fim de efetuar a transferência de suas PDU's (CMIPDU's).

Um protocolo de suporte para comunicações de gerenciamento, deve transferir PDU's de operações e notificações de gerenciamento e deve também fornecer suporte para controle de acesso da MIB [8]. Além dos objetos da MIB, o controle de acesso se estende a todas informações contidas nos relatórios de eventos emitidos pelo agente. As funcionalidades de controle de acesso dos objetos gerenciados estão definidas em [13]. Relacionado aos relatórios de eventos, suas informações são controladas de acordo com o comportamento definido para o discriminador de repasse de eventos [14]. Outras características adicionais, dentre elas, a seleção de objetos por escopo e filtro, devem também ser suportadas para fins de gerenciamento. O CMIP pode ser utilizado em ambientes que implementam toda a pilha de protocolos OSI ou também pode-se utilizá-lo sobre TCP/IP [15] sendo esta, uma arquitetura alternativa chamada CMOT [8].

A Figura 2.6 apresenta a camada de aplicação com os ASE's utilizados por aplicações de gerência.

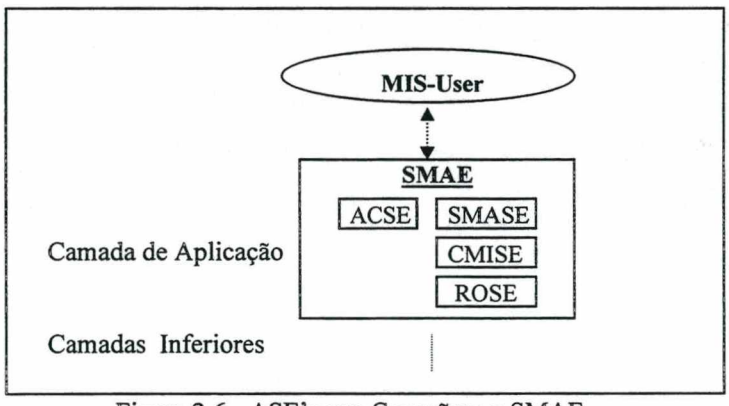

Figura 2.6 - ASE's que Compõem a SMAE.

[1] A sigla MIS nesta referência, coincide com MIS (*Management Information Service*) definida nos documentos de padronização do gerenciamento OSI/ISO.

2.5 - Áreas Funcionais

A definição de gerência de sistemas proposta pela ISO, em seu modelo OSI, abrange vários aspectos funcionais relacionados aos sistemas abertos. A fim de organizar as diferentes áreas de atuação do modelo, foram definidas cinco áreas funcionais específicas (SMFA) (Falta, Segurança, Performance, Contabilização, Configuração), agrupando em cada uma, propostas de gerenciamento individuais.

Para cada uma destas áreas, foram definidas várias funções de gerência, sendo que algumas delas servem a mais de uma área funcional. Um exemplo de função compartilhada por várias áreas funcionais é a função de gerenciamento de teste (TMF), a qual está presente nas cinco áreas funcionais [8]. Ao contrário da TMF, a função de relatório de alarme de segurança (SARF) somente é utilizada para fins de gerenciamento de segurança.

2.6 - Aspecto Organizacional do Modelo

O aspecto organizacional do modelo descreve a natureza distribuída do gerenciamento OSI [3]. A fim de controlar e monitorar uma coleção de objetos gerenciados, os requisitos organizacionais abrangem:

- Particionar o ambiente de gerenciamento de acordo com funcionalidades, como descrito na seção 2.5, ou de acordo com outras propostas de gerenciamento como estruturas geográficas, tecnológicas, etc. [3];
- Aplicações de gerência devem assumir papeis de gerente ou agente, dependendo da proposta estabelecida para o domínio em uma determinada associação de gerenciamento;
- Exercer de forma consistente as várias políticas [16] de gerenciamento.

Organizando os objetos gerenciados de maneira a respeitar os requisitos descritos anteriormente, este conjunto de objetos é chamado de domínio de gerenciamento. A Figura 2.7 ilustra o conceito de domínios de gerenciamento.

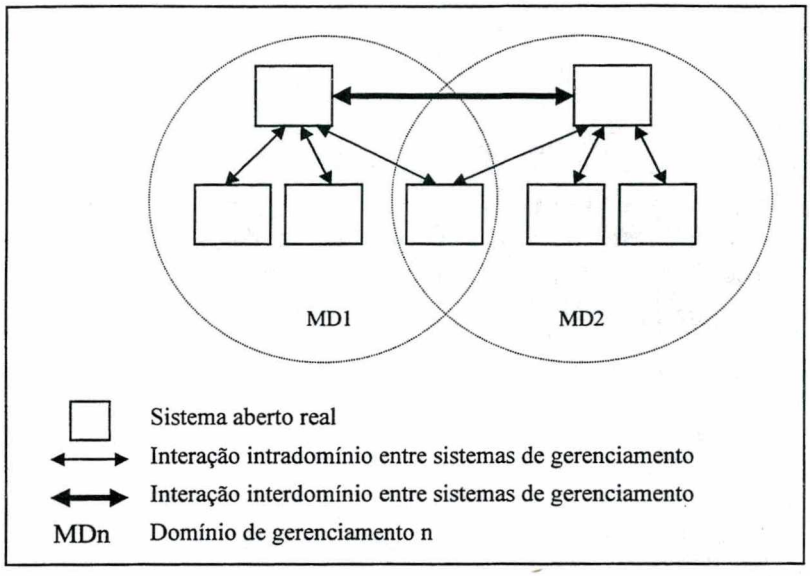

Figura 2.7 - Conceito de Domínios de Gerenciamento [3].

CAPÍTULO III

Projeto: Plataforma de Gerenciamento OSI

3.1 - Introdução

Atualmente, é objeto de vários projetos de pesquisa [17][18], a especificação e implementação de plataformas para gerenciamento de redes, com o objetivo de fornecer um gerenciamento integrado em ambientes heterogêneos.

A semântica de heterogeneidade em gerenciamento de sistemas, está relacionada com o uso de informações diferentes, por diferentes dispositivos, para representar comportamentos de rede similares [19].

Várias ferramentas de apoio ao gerenciamento de redes são comercializadas, sendo a maioria construída para gerenciar dispositivos e *softwares* de um determinado fabricante (IBM™, HP™, etc.).

Dois modelos são largamente adotados para a construção de plataformas de gerenciamento de redes. Um baseado no protocolo CMIP (padrão *de jure)*, e outro baseado no protocolo SNMP (padrão *de facto*) [8]. No mundo *Internet*, redes baseadas em TCP/IP, a utilização do protocolo SNMP é predominante, em função de sua implementação simples, podendo ser embutida (*embedded*) em vários dispositivos de redes (roteadores, *hubs*, *switches*, etc.).

O CMIP por ser mais robusto, em função de sua maior complexidade e funcionalidade, exige um maior esforço em sua implementação, consumindo maiores recursos (memória, CPU, etc.) para seu processamento.

Atualmente, os maiores fornecedores (IBM, HP, Sun™, Cabletron, etc.) de produtos para gerenciamento de redes, oferecem suporte ao CMIP em seus produtos, visto ser uma padronização internacional e oferecer várias funcionalidades não apresentadas pelo SNMP. No mundo TMN (*Telecommunication Management Network*) [8], a exigência de produtos com suporte CMIP é hoje uma realidade, sendo uma exigência do mercado para com os fornecedores de *software* e *hardware* de

telecomunicações. Várias plataformas do mercado já estão implementando *gateways* para ambos protocolos, o que as permite interoperar nos dois ambientes.

O objetivo em se utilizar protocolos abertos para gerenciar redes de comunicação é reduzir o custo do gerenciamento destas redes, fornecendo interfaces padronizadas dentre as diferentes tecnologias e serviços existentes, aumentando a interoperabilidade em ambientes multifornecedores [20].

3.2 - Uma Visão Geral do Projeto

O grupo de redes do Laboratório de Integração de *Software* e *Hardware* (LISHA) da Universidade Federal de Santa Catarina (UFSC) está desenvolvendo um projeto que visa a implementação de uma plataforma para gerência de redes, a qual segue o modelo de gerenciamento OSI/ISO.

Este projeto tem como pesquisadores, alunos do curso de graduação e pós-graduação em Ciência da Computação da Universidade Federal de Santa Catarina, sendo objeto de vários trabalhos de conclusão de curso, dissertações de mestrado e publicações em congressos nacionais e internacionais. A Figura 3.1 apresenta a estrutura da plataforma.

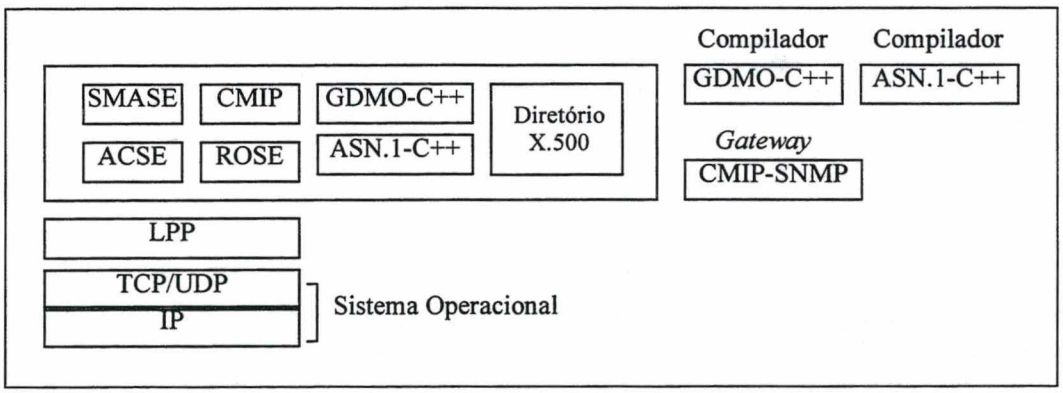

Figura 3.1 - Estrutura da Plataforma.

Como ilustrado na Figura 3.1, a plataforma é composta de vários módulos que encapsulam funcionalidades específicas, dentre os quais alguns são particularmente utilizados para fins de gerenciamento e outros genéricos, utilizados em várias aplicações do ambiente OSI, como exemplo o ACSE.

Como a plataforma tem por objetivo inicial atuar em um ambiente de rede TCP/IP, ambiente da rede UFSC, foi utilizada a implementação da pilha TCP/IP

fornecida pelo próprio sistema operacional, o qual a implementa ao nível de *kernel*. O Sistema operacional utilizado é o SunOs™ 5.4 (Unix Solaris™ 2.4).

A linguagem de programação utilizada na implementação da plataforma é a linguagem C++. A construção das interfaces gráficas para as aplicações gerente, está sendo realizada para ambiente OpenWindows. Todos os programas que implementam as interfaces seguem o padrão OpenLook. Nesta fase, todo ambiente de desenvolvimento está baseado no UNIX, contudo, é objeto de estudos futuros, o porte destas aplicações para outros ambientes operacionais.

3.2.1 - Suporte GDMO/ASN.1 - C++

Primeiramente, foi identificada a necessidade de se especificar regras de mapeamento dos *templates* GDMO para a linguagem C++, o que permite a implementação das especificações dos objetos gerenciados em uma linguagem de programação. Após esta fase, tais regras foram então implementadas juntamente com vários tipos ASN.1, visto que as sintaxes dos tipos de dados definidos nos *templates* GDMO estão em notação ASN.1.

A implementação dos tipos ASN.1 simples (ex. INTEGER) e construtores (ex. SEQUENCE) foi realizada em [21].

Para cada tipo ASN.1, tem-se uma classe de objetos C++ correspondente. Todas as classes de tipos ASN.1, são derivadas de uma classe base virtual chamada Attr_Types [21]. Nesta classe, estão definições de métodos virtuais os quais devem estar presentes em todas suas classes derivadas, ou seja, classes de tipos ASN.1.

Quando o usuário define um tipo de atributo em GDMO, o mesmo será incluído nesta hierarquia, sendo derivado de seu tipo ASN.1 primitivo. Por exemplo, o usuário definindo a classe de objeto gerenciado *hub*, especifica um atributo do tipo números DePortas, o qual possui como sintaxe ASN.1 o tipo simples INTEGER. A classe C++ números DePortas será derivada da classe C++ INTEGER, sendo que atributos do tipo números DePortas, serão objetos desta classe. Para os tipos construtores, foram criadas estruturas de dados genéricas, as quais armazenam tanto tipos simples quanto compostos. Um exemplo é a classe SEQUENCE, a qual pode conter objetos representando tipos simples (ex. REAL), como também outros objetos da classe SEQUENCE, visto a natureza recursiva dos tipos construtores.

Ainda relacionado ao processo de mapeamento GDMO/ASN.1 - C++, está em andamento um projeto que visa a construção de compiladores para a automatização destes dois processos. A Figura 3.2 apresenta o protótipo do compilador GDMO-C++.

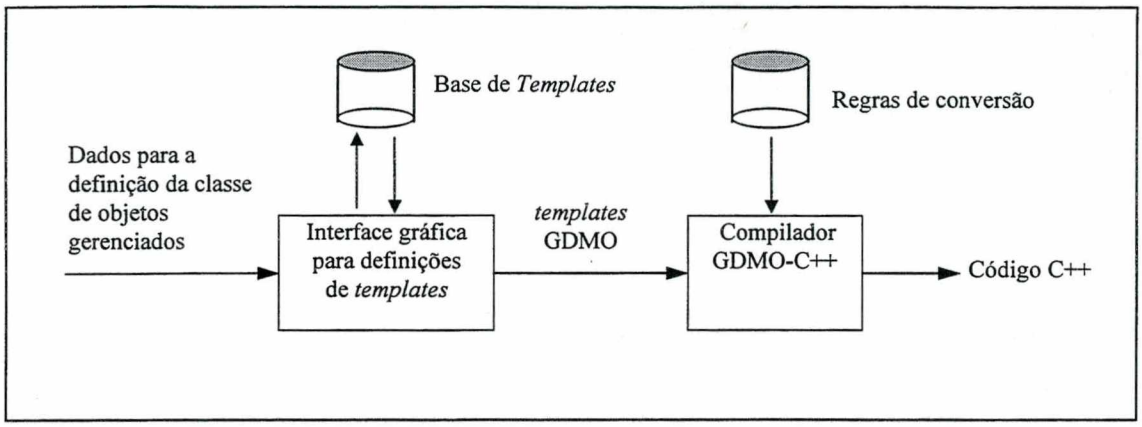

Figura 3.2 - Visão Geral do Protótipo para o Compilador GDMO-C++.

As regras de conversão que aparecem na Figura 3.2, alimentando o compilador GDMO-C++, são aquelas definidas em [21]. Além do compilador, está sendo construído um ambiente para a definição de *templates* GDMO, o qual servirá de interface entre o projetista da aplicação de gerenciamento e o compilador. Para o desenvolvimento deste ambiente, está sendo realizado todo trabalho de definição de bases de dados e relacionamento destas, a fim de se criar um conjunto de bases de informações a respeito dos vários *templates* (notificação, pacotes, atributos, etc.) e tipos de dados que envolvem a modelagem de objetos gerenciados.

3.2.2 - LPP

A implementação de algumas funções do LPP (regras de codificação e interface com a camada de transporte) foram realizadas [22]; contudo, somente foram concluídas aquelas relativas à interface com a camada de transporte e as funções de conversão de sintaxe interna para sintaxe de transferência dos tipos simples, implementados no trabalho de mapeamento GDMO-C++. As funções de codificação (*encode*) e decodificação (*decode*) dos tipos ASN.1, servem de suporte à implementação das classes geradas pelo compilador GDMO/ASN.1-C++.

3.2.3 - Funções de Gerenciamento de Sistemas

Foram implementadas as funções de relatório de eventos e relatório de alarme de segurança [23], com seus respectivos objetos de suporte (discriminador, discriminador de repasse de eventos, etc.) e sua interface gráfica para o gerente. Esta interface permite configurar e visualizar os relatórios de eventos e alarmes de segurança emitidos pelo agente. A função de relatórios de eventos é uma funcionalidade básica e, para fins de teste das aplicações de gerenciamento, terá papel importante. Esta função permite, ao processo gerente, controlar a emissão de relatórios emitidos pelo agente, especificando dentre outras coisas, horários e destinos para tais relatórios. Utilizando os serviços desta função, a função de relatório de alarme de segurança será utilizada para criar aplicações agentes que atuem na área funcional de segurança.

3.2.4 - Gateway CMIP-SNMP

Em [24][25] foram realizados trabalhos de definição e especificação formal de um *gateway* CMIP-SNMP. O objetivo do *gateway* é permitir que o gerente da plataforma OSI, possa interagir com agentes que seguem tanto o protocolo CMIP quanto SNMP. Este *gateway* foi implementado a nível de operações de gerenciamento. Caso a associação realizada seja com um agente SNMP, todas operações de gerenciamento para este agente serão convertidas para o protocolo SNMP e transferidas, seguindo os padrões definidos para o SNMP. Na recepção de respostas ou *traps*, o processo de conversão também deve ser realizado, pois o gerente sendo CMIP, não entende PDUs SNMP. Para algumas funcionalidades existentes no CMIP e ausentes no SNMP, como escopo e filtro, foram inseridos processos adicionais a fim de permitir que estas funcionalidades sejam executadas no lado SNMP, como se estivessem sendo submetidas a um agente CMIP.

3.2.5 - ACSE, ROSE, CMIP e X500: Futuros Projetos

A prioridade para os próximos projetos serão as conclusões dos módulos ACSE e ROSE, os quais foram iniciados em [26] e a implementação dos módulos CMIP e X500 (serviço de diretório), sendo também alvo de futuros trabalhos a implementação das várias funções de gerência disponíveis no SMASE. Atualmente,

os serviços oferecidos por estes módulos, estão sendo simulados pelos códigos já implementados. Foi definido que aplicações gerente não consultam o serviço de diretório para obter informações a respeito do endereço de um determinado agente, sendo esta consulta simulada por definir o endereço do agente em arquivos locais ao gerente. Outras funcionalidades, que ainda não estão disponíveis, serão possíveis com a implementação destes módulos. Um estudo dos possíveis serviços do X.500 que, podem ser utilizados para fins de gerenciamento, já está em andamento.

3.2.6 - Construção de Agentes na Plataforma

Para a construção de agentes, deve se ter definido um núcleo básico de operação para tais aplicações. Desta forma, a complexidade na implementação dos agentes é reduzida, pois todo suporte à sua execução é oferecido e controlado pelo núcleo, não sendo responsabilidade do implementador da aplicação agente.

Atualmente, as principais plataformas de gerenciamento disponíveis no mercado, oferecem *Agent Toolkits* (ATKs) para a automatização do processo de criação de agentes. Estes ATKs geram automaticamente todo código que implementa o núcleo do agente, ficando a cargo do usuário, implementações de mais alto nível, relacionadas ao comportamento dos objetos gerenciados.

No projeto da plataforma, não se tem definido nenhum núcleo para os processos agentes.

Este trabalho, visa a definição e implementação de um núcleo para agentes da plataforma, possibilitando a futura criação de um ATK para a automatização do processo de desenvolvimento de agentes.

Os próximos capítulos investigarão os requisitos e funcionalidades dos processos agentes de gerenciamento OSI, juntamente com conceitos de *multithreading*, os quais oferecerão informações para a posterior apresentação da proposta e sua implementação.

CAPÍTULO IV

Agentes de Gerenciamento OSI

4.1 - Introdução

Os documentos de padronizações não definem a organização interna dos processos de gerenciamento (Agentes/Gerentes), sendo esta uma questão local de cada implementação.

A fim de se implementar tais processos, torna-se necessário identificar seus requisitos e características, propondo uma estrutura na qual os elementos de serviços e protocolos, disponíveis para as aplicações de gerenciamento, possam ser utilizados. O restante deste capítulo será dedicado à análise dos requisitos e características para a implementação de aplicações de gerenciamento.

4.2 - Conceito de Aplicação de Gerenciamento (MIS-Users)

Conceitualmente, processos de aplicação de gerenciamento são aquelas aplicações que se utilizam dos serviços providos pelo elemento de serviço de aplicação de gerenciamento de sistemas (SMAE). Este conceito, definido em [3], pode ser ilustrado na Figura 4.1.

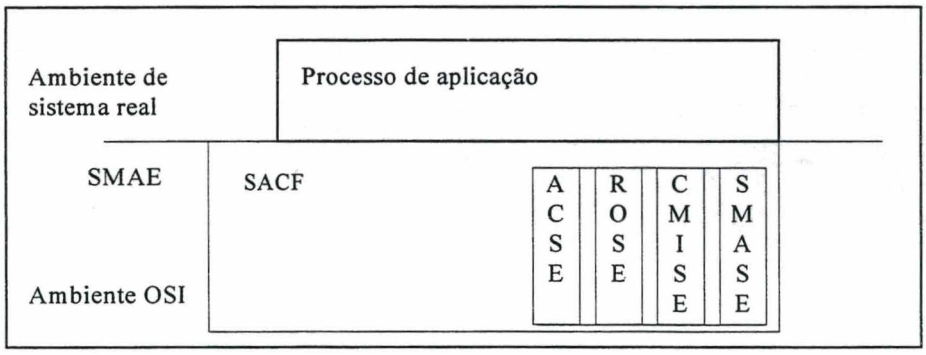

Figura 4.1 - Gerenciamento e a Camada de Aplicação [3].

Processos de aplicação são parte do ambiente real, não sendo escopo dos documentos de padronização, os quais somente tratam de componentes definidos dentro do ambiente OSI (elementos de serviços, protocolos de camada, etc.).

De forma genérica, processos de aplicação de gerenciamento podem assumir tanto o papel de gerente (*Manager Rule*), quanto o papel de agente (*Agent role*).

Este trabalho está envolvido com processos de aplicação atuando no papel de agente. A seguir, processos agentes serão abordados em detalhes.

4.3 - Processos Agentes

Agentes são processos participantes do gerenciamento de sistemas, que têm como objetivo servir e manter informações de gerenciamento a respeito de um determinado sistema gerenciado.

Para atender a estes requisitos, agentes devem monitorar e atuar sobre recursos do sistema. Monitoração pode ser entendida como a coleta dinâmica de informações relacionadas com os objetos, de *software* ou *hardware*, que estão sob observação [27]. Controle pode ser definido, como a execução das operações requisitadas pelo gerente sobre os objetos gerenciados.

Baseado nos conceitos de monitoração e controle, definidos anteriormente, o processo agente deve ser estruturado de forma a minimizar os tempos de respostas às requisições dos gerentes e manter sua base de informações de gerenciamento (MIB) sempre atualizada e consistente.

A seguir foram identificados dois requisitos básicos que um processo de aplicação agente deve suportar, são eles:

- Suportar uma abordagem orientada a eventos;
- Suportar a execução de *polling*, em tempo real, para interações com recursos reais do sistema.

Em função das aplicações agentes necessitarem de um suporte para comunicações externas (ambiente distribuído), estas são organizadas seguindo um enfoque orientado a eventos. Os tipos de comunicações que aplicações agentes estão sujeitas, podem ser identificadas a partir de suas possíveis interações:

- Gerente $\Rightarrow$ Agente (1)
- Agente $\Rightarrow$ Gerente (2)
- Agente $\Rightarrow$ Recurso Gerenciado (3)

A primeira interação é necessária em função das requisições de operações de gerenciamento, realizadas pelo gerente, sobre um sistema gerenciado (Agente).

A segunda interação, pode ser tanto para a emissão de respostas às operações requisitadas (1ª interação), quanto para a emissão de relatórios de eventos ao sistema gerente.

Para a terceira interação, requisitos de comunicações externas dependem principalmente da natureza do recurso gerenciado. Comunicações externas são exigidas, quando se tem recursos gerenciados os quais estão fracamente acoplados ao sistema agente. Um exemplo de recurso fracamente acoplado pode ser uma impressora que esteja conectada diretamente à rede e que deve ser gerenciada por um agente localizado em uma estação de trabalho.

Neste caso, a comunicação agente-impressora é necessária, através de uma interação não padronizada que deve ser feita de maneira transparente para o processo gerente. Esta comunicação é totalmente dependente da interface do recurso (impressora).

Recursos fortemente acoplados são aqueles que podem ser acessados no mesmo espaço do processo agente, ou seja, dentro do mesmo sistema local (variáveis do *kernel*, entidades de camada, etc.).

Processos sendo gerenciados em sistemas que implementam proteção de memória, por exemplo UNIX, são considerados recursos fracamente acoplados, e interagem com o processo agente através de algum mecanismo de IPC (*Interprocess Communication*).

A interação com os recursos reais, estejam eles fortemente ou fracamente acoplados, é um ponto extremamente importante, o qual deve ser analisado cuidadosamente no estudo e implementação de aplicações agentes. Toda organização das atividades internas dos agentes deve ser estruturada a fim de suportar, de forma segura e eficiente, estas formas de interação.

A necessidade dos objetos gerenciados de interagirem com os recursos reais, exige que processos agentes ofereçam uma estrutura que suporte a execução de *polling* sobre estes recursos. O acesso periódico aos recursos monitorados, com o objetivo de emitir notificações ao gerente, e se manter atualizado com relação ao estado do recurso sendo representado, é essencial para o gerenciamento de recursos do mundo real (ex. *modem*, hubs, conexões, etc.).

Existem recursos que possuem capacidades de emitir alarmes mediante a ocorrência de algum evento previamente programado, estes são freqüentemente chamados dispositivos inteligentes (*clever*) [28] e trazem embutidas funções de gerência que facilitam a interação com os objetos gerenciados.

Os requisitos de tempo real nas operações de *polling* são desejáveis, pois o objeto gerenciado deve representar o recurso real de maneira consistente, a fim de servir corretamente às requisições do gerente, mantendo desta forma a MIB sempre atualizada.

O processo de interação com o recurso, é parte do comportamento (*behaviour*) do objeto, sendo implementado pelo desenvolvedor da aplicação agente. Nem sempre é necessário que objetos gerenciados fiquem executando *polling* sobre os recursos representados, sendo uma questão definida no comportamento do objeto.

Existem três possíveis tipos de interação entre o objeto gerenciado e o recurso real sendo representado.

- Acesso por demanda (*access-upon-external-request*);
- *Cache* através de *polling* periódicos;
- Recepção de eventos assincronamente.

A primeira interação significa que nenhuma atividade é realizada pelo agente enquanto este não recebe uma solicitação do gerente. Neste caso, não existe o suporte à emissão de notificações mediante eventos internos e/ou externos, a não ser aqueles iniciados por um acesso do gerente (Ex. notificação de deleção de objeto).

No segundo caso, as respostas às requisições são mais rápidas, pois não exigem o acesso ao recurso, visto que as informações estão armazenadas em *cache*. Esta forma de interação não se aplica a recursos que exigem uma coleta no momento

da requisição, pois a informação contida no *cache* pode estar, naquele momento da requisição, desatualizada (*timeliness*) com o estado do recurso real.

A última forma de interação é interessante, mas somente se esta garantir que não exista *overhead* desnecessário quando o agente não está atendendo requisições do gerente.

Relacionado à primeira interação, os requisitos de representação em tempo real dos recursos gerenciados são mínimos. Um exemplo de recurso com acesso por demanda, pode ser um objeto gerenciado que representa a quantidade de memória disponível no sistema em um dado instante. Pelas características do recurso que se pretende gerenciar, o comportamento do objeto não exige a execução de *polling*, pois a informação que o gerente necessita deve ser coletada somente no instante da requisição da operação (Ex. M-GET).

Caso o objeto gerenciado fosse modelado de forma a suportar a emissão de notificações, quando a memória do sistema fosse menor que um valor (*threshold*) especificado, seria então necessária a monitoração freqüente do recurso real (memória) a fim de atender aos requisitos de emissão de notificações.

Como pode ser observado, o tipo de interação entre o objeto gerenciado e o recurso que ele representa, é definido pelo projetista do objeto, de acordo com os seus requisitos de gerenciamento para um determinado recurso.

Todos os tipos de interações entre objetos gerenciados e recursos reais, citados anteriormente, devem ser suportados pelas aplicações de gerenciamento atuando no papel de agente.

Vale ressaltar, que no exemplo do recurso gerenciado "memória do sistema", este possui a característica de estar fortemente acoplado ao sistema, facilitando a sua interação com o objeto gerenciado. É freqüente se ver agentes controlando informações como o tempo em que o sistema está no ar (*sysUptime*), configurações de roteamento, taxa de utilização da CPU, número de colisões na interface de rede, etc., isto porque todas estas informações estão no ambiente local do processo agente, sendo a maioria disponível como variáveis do sistema operacional, o que facilita sua utilização.

Já recursos fracamente acoplados, exigem maiores esforços para sua representação em forma de objetos gerenciados. Quando tais recursos possuem uma

interface que possa ser usada a fim de coletar informações e alterar seu comportamento, a implementação dos objetos gerenciados é facilitada. Um exemplo pode ser a interface serial dos *modems*, que na maioria das vezes traz alguns sinais de controle habilitados, a fim de fornecer informações a respeito do seu estado.

Recursos que estão fracamente acoplados e não oferecem informações através de interfaces, são difíceis de serem representados, sendo que para alguns casos, não existem formas de representação através de objetos gerenciados.

É responsabilidade do núcleo do agente, oferecer as funcionalidades que suportem os serviços requeridos pelas especificações do comportamento do objeto, realizadas na modelagem da informação de gerenciamento.

4.4 - Núcleo de Agentes

Baseado nas definições contidas nos documentos de padronização e nos componentes identificados como sendo essenciais para sua execução, agentes são basicamente compostos de dois módulos:

- Módulo de serviços e protocolo de gerenciamento;
- Núcleo básico de operação.

O primeiro permite ao processo agente interagir com os elementos de serviços padronizados da camada de aplicação (ACSE, ROSE, CMISE, etc.).

O segundo componente, implementa as funcionalidades específicas aos processos agentes. São exemplos de tais funcionalidades:

- Estrutura para implementação da MIT, juntamente com seu gerenciamento;
- Suporte às instâncias das classes de objetos gerenciados (MIB);
- Mecanismos de distribuição das operações submetidas aos objetos da MIB;
- Implementação das várias funções de gerenciamento, objetos de suporte e funcionalidades específicas (ex. persistência p/ os objetos da MIB);

- Oferecer APIs de alto nível para acesso dos serviços do núcleo (itens anteriores).

As funcionalidades acima, apenas listam os requisitos básicos para a execução dos processos agentes. Características adicionais (Ex. performance, tolerância a faltas, etc.) são possíveis, a fim de incrementar os serviços oferecidos.

4.4.1 - Estado da Arte

Atualmente, a maioria das implementação de agentes CMIP seguem uma abordagem orientada a *callbacks*. A utilização de *callbacks* é freqüente em aplicações orientadas a evento.

Callbacks é um idioma comum na programação orientada a eventos, onde um cliente fornece uma função (função de *callback*) para um servidor, e este por sua vez invoca esta função quando uma apropriada condição seja satisfeita (Ex. conclusão de um serviço requisitado pelo cliente). Sistemas para interface com usuário (Ex. *X Window System*), aplicações de controle e automação de processos, dentre outros, são alguns exemplos de aplicações estruturadas de forma orientada a eventos.

Seguindo esta filosofia, os núcleos atuais são divididos essencialmente em duas partes:

- Serviços CMIS genéricos;
- Suporte à *callbacks*.

Relacionado aos serviços CMIS genéricos, destacam-se todas as funções relacionadas a recepção e envio de mensagens CMIP, manutenção da árvore de informação de gerenciamento (MIT), mecanismos de acesso aos objetos, funções de gerenciamento, dentre outras. A Figura 4.2 ilustra esta arquitetura.

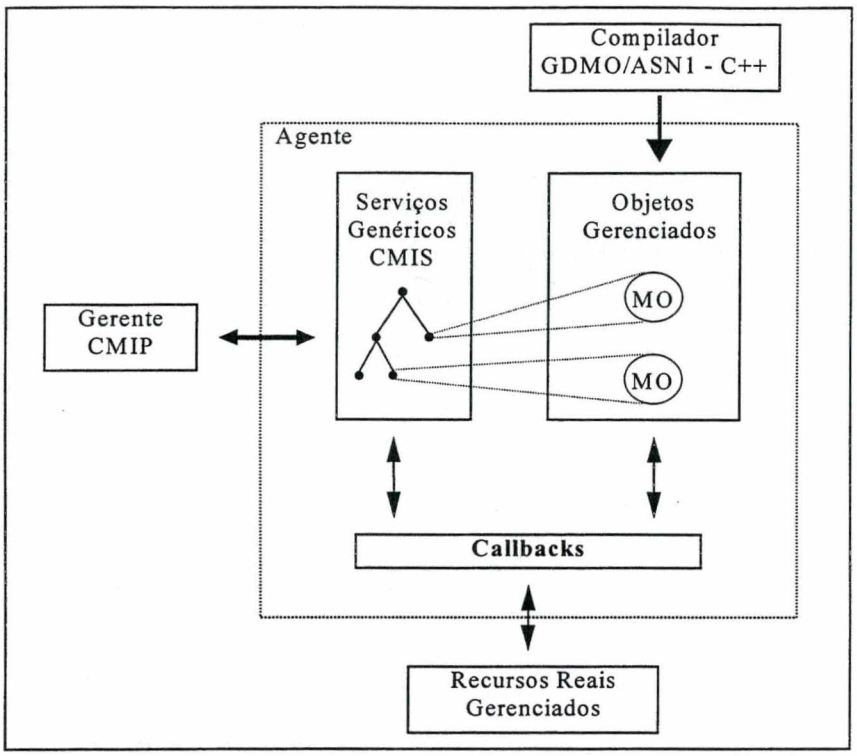

Figura 4.2 - Arquitetura de Agentes seguindo abordagem de Callbacks.

Os mecanismos de acesso aos objetos, ativam os códigos que tratam detalhes de execução dos comandos CMIP, na fronteira das instâncias das classes de objetos gerenciados. O código referente a estes mecanismos, é gerado a partir das especificações do mapeamento GDMO/C++.

O acesso ao objeto pode ser tanto síncrono como assíncrono, dependendo da implementação do núcleo.

O código de execução das *callbacks* deve ser implementado manualmente, não sendo gerado pelo compilador GDMO/C++. Este processo é bastante difícil de se automatizar, visto que para cada recurso gerenciado, existe uma forma proprietária de interação.

No processo de geração do núcleo, um corpo (*skeleton*) para as várias *callbacks* é criado, juntamente com seus comentários extraídos a partir dos documentos GDMO. Muitos destes comentários são mapeamentos diretos das definições do comportamento (*Template* BEHAVIOUR) do objeto. O objetivo dos comentários é orientar o implementador da aplicação agente sobre o que deve ser implementado para cada uma das rotinas de *callbacks*.

Relacionado a comunicação com as aplicações gerentes, atualmente são utilizadas as interfaces XOM/XMP [29].

Na implementação de agentes CMIP, o modelo de *callbacks* é o mais adotado pelas principais plataformas de gerenciamento disponíveis atualmente. Quando uma requisição do gerente chega ao processo agente, este realiza o processamento padrão (verificação da semântica dos parâmetros da operação, controle de acesso, etc.), efetua a localização do objeto ao qual se destina a requisição, e posteriormente invoca a *callback* que implementa aquela operação. Esta invocação pode ser efetuada de forma síncrona e/ou assíncrona, dependendo de como o núcleo implementa este serviço. Após a realização da operação, a *callback* retorna um resultado de erro ou sucesso, o qual será enviado ao processo gerente.

Vale ressaltar, que a forma como são realizadas as interações dos objetos gerenciados com os recursos reais, invocação das *callbacks* e retorno destas invocações, todos estes fatores irão limitar ou não o tipo de interação entre o gerente e o processo agente. Na seção 6.9 uma abordagem destas limitações será apresentada.

A seguir, será realizado um estudo de casos de duas implementações de núcleos de agentes, uma utilizada pela plataforma Solstice (*Solstice TMN Agent Toolkit*) da Sun Microsystems, e a outra utilizada na plataforma OSIMIS da UCL (*University College London*).

4.4.2 - Estudo de Casos

Nesta seção, serão apresentadas duas implementações para núcleos de agentes. Inicialmente será apresentado o núcleo gerado pelo Solstice TMN ATK, e posteriormente o núcleo de agentes oferecido pela plataforma OSIMIS.

O Solstice TMN ATK [30] é um ambiente completo para desenvolvimento de Agentes CMIP, sendo criado para a implementação de agentes Q3 [8] em plataformas TMN. Este núcleo segue o modelo apresentado na seção anterior, sendo portanto orientado a *callbacks*.

O segundo núcleo apresentado, faz parte da plataforma OSIMIS. Este segue uma arquitetura um pouco diferente da utilizada no Solstice TMN ATK, sendo uma variante do modelo de *callbacks*.

4.4.2.1 - Núcleo de Agentes no Solstice TMN ATK

O Solstice TMN ATK é uma ferramenta comercial para desenvolvimento de agentes. Esta provê um ambiente para desenvolvimento de agentes CMIP (TMN/Q3), incluindo um núcleo para o agente, métodos para comunicação agente/objeto e APIs padronizadas para interações com gerentes CMIP.

Agentes desenvolvidos com o Solstice TMN ATK podem ser executados em plataformas SPARC ou x86 [30].

O processo de desenvolvimento é basicamente o mesmo apresentado na seção 4.4.1. Um compilador GDMO recebe como entrada arquivos GDMO padrão, processa estas definições GDMO, e gera como saída o código correspondente aos esqueletos das *callbacks*, as quais são ligadas ao núcleo do agente.

O Núcleo manipula todas operações CMIP, realização de *scoping* e filtro, operações de *linked replies*, manipulação de eventos, e oferece um acesso síncrono ou assíncrono para a manipulação da MIT.

As APIs oferecidas ao desenvolvedor são:

- *Control* API - Serve para inicialização e gerenciamento do núcleo;
- GO (*Generic Object*) API - Esta API oculta os detalhes de manipulação do XOM/XMP, ajudando na implementação dos objetos;
- MIT API - Controle da MIT;
- DMI (*Definition of Management Information*) API - Oferece uma interface de alto nível para a manipulação dos objetos padrões definidos pela DMI [31];
- *Response* API - Utiliza como suporte a LL API, a fim de responder as requisições e realizar a emissão de notificações;
- LL (*Low Level*) API - Suporta o uso dos modos de operação síncrono e assíncrono (Serve de suporte principalmente para os módulos GO e *Response* APIs);

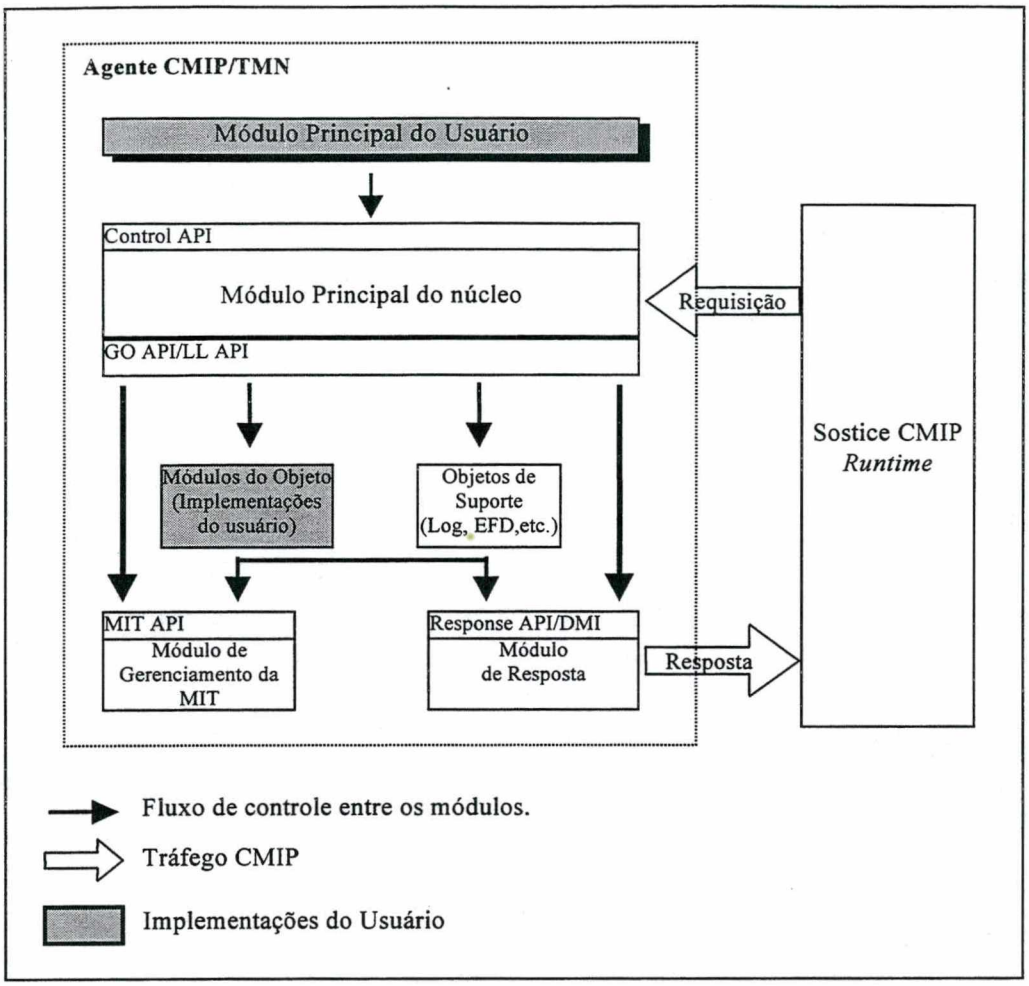

Figura 4.3 - Fluxo de Controle dentro do Agente Solstice TMN.

As funções de gerenciamento e objetos de suporte implementados são:

- OMF (*Object Management Function*) (X.730/ISO 10165-1)
- SMF (*State Management Function*) (X.731/ISO 10165-2)
- ATRR (*Attributes for Representation Relationships*) (X.732/ISO 10165-3)
- ARF (*Alarm Reporting Function*) (X.733/ISO 10165-4)
- EFD (*Event Forwarding Discriminator*) (X.734/ISO 10165-5)

A Figura 4.3 apresenta o relacionamento destes componentes dentro do processo agente.

Neste modelo, a interação com os recursos reais segue um regime de acesso por demanda (*access-upon-external-request*). O núcleo é preparado para não realizar atividades quando não se tem requisições do gerente.

Nos casos em que o comportamento do objeto exige a freqüente interação com o recurso (*polling* periódicos), a modificação do código gerado pode ser realizada pelo desenvolvedor da aplicação agente, a fim de suportar tal característica. Esta necessidade de modificação realizada pelo usuário, é uma limitação do modelo.

O desenvolvimento do processo agente, utilizando o Solstice TMN ATK, está ilustrado na Figura 4.4.

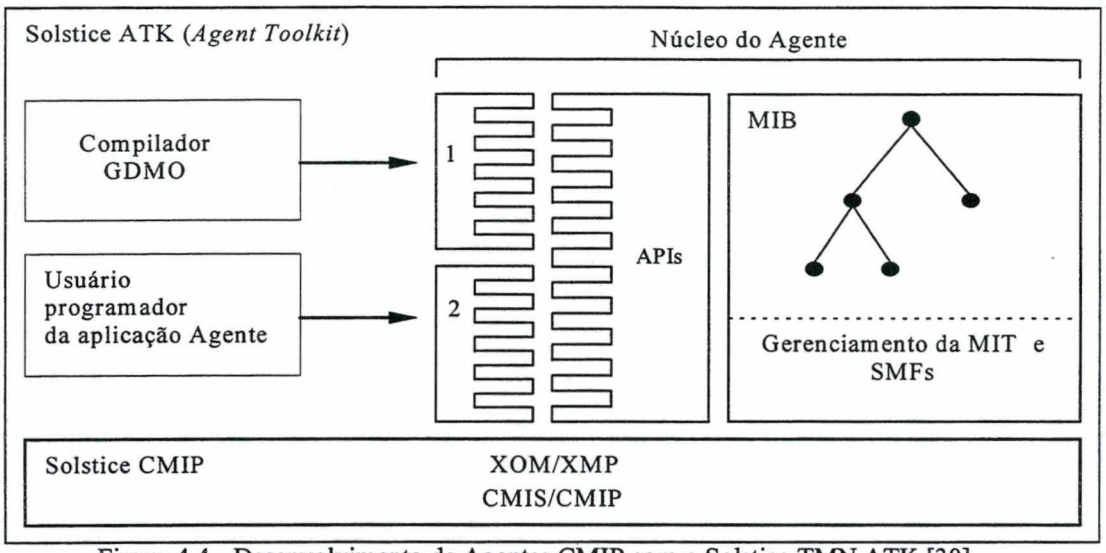

Figura 4.4 - Desenvolvimento de Agentes CMIP com o Solstice TMN ATK [30].

Somente o código representado pela caixa 2, o qual define detalhes específicos da interação entre o recurso real gerenciado e o objeto que o representa, deve ser gerado manualmente pelo desenvolvedor da aplicação agente. Todos os outros componentes são fornecidos pela plataforma na geração do código apartir das definições GDMO.

4.4.2.2 -Núcleo de Agentes no OSIMIS

OSIMIS [28] é uma plataforma de gerenciamento orientada a objetos que se baseia no modelo de gerenciamento OSI. Sua implementação é basicamente realizada na linguagem C++.

Uma das principais características do OSIMIS, o que o torna largamente utilizado em pesquisas de gerência de redes no meio acadêmico, são suas interfaces de alto nível para o desenvolvimento de aplicações de gerenciamento, as quais ocultam as complexidades impostas pelo modelo de gerenciamento OSI (Ex. manipulação de primitivas CMIP, MIT, etc.).

A plataforma OSIMIS utiliza o ISODE (*ISO Development Environment*) [32] como suporte básico para comunicação (*Stack OSI*) mas também pode ser adaptada para utilizar as APIs de gerenciamento XOM/XMP, o que permite a utilização de uma larga faixa de produtos disponíveis atualmente.

Referente aos agentes, o OSIMIS oferece os seguintes componentes:

- Compiladores GDMO/ASN.1;
- Mecanismo de Coordenação que permite estruturar a aplicação em um modelo totalmente orientado a eventos;
- GMS (*Generic Managed System*) o qual é um *engine* (suporte) orientado a objetos para a criação de agentes OSI, oferecendo uma API de alto nível para implementação de classes de objetos gerenciados e funções de gerenciamento.

Programadores de aplicações de gerenciamento, utilizam as interfaces acima a fim de implementarem seus agentes sobre o OSIMIS. Todas requisições e respostas são baseadas em chamadas de procedimentos. As primitivas *indication* e *confirm* são implementadas através de uma única chamada "*wait*" [28].

Em função das necessidades de suportar o tratamento de eventos internos e externos de comunicação (um dos requisitos das aplicações no papel de agente [seção 4.3]), o OSIMIS oferece um mecanismo de coordenação de eventos orientado a objetos. Desta forma, todos os eventos internos ou externos são serializados seguindo uma política "FCFS (*First-Come-First-Served*)". Este mecanismo permite a fácil integração de fontes de eventos internas ou externas ao núcleo do agente.

O GMS (*Generic Managed System*) oferece os serviços para a construção de agentes, suportando as funcionalidades definidas no modelo de gerenciamento OSI. Este oferece diferentes métodos de acesso para os recursos reais gerenciados, incluindo mecanismos de procuração (*proxy*), sendo suportado pelo mecanismo de coordenação de eventos.

O GMS é composto basicamente por três classes que se interagem [17]:

- CMISAgent - Oferece facilidades de agente;
- MO - Oferece suporte aos objetos gerenciados;
- MOClassinfo - Uma Meta Classe para as classes de objetos gerenciados.

O CMISAgent tem a função de aceitar as associações de gerenciamento, realizar validações de controle de acesso, verificar a validade dos parâmetros das operações e executar as sincronizações nas operações com requisitos de sincronismo.

Atualmente, a comunicação do objeto CMISAgent com os objetos gerenciados (MO) é realizada somente na forma síncrona. Uma invocação de um método do objeto gerenciado deve sempre retornar um resultado válido ou erro. Isto significa, que objetos gerenciados que representam recursos reais fracamente acoplados e que se comunicam com estes sobre um regime de acesso por demanda (*access-upon-external-request*), terão sua comunicação com o recurso real realizada de forma síncrona, resultando no bloqueio da aplicação até a recepção dos resultados da requisição. Isto é um problema se outras requisições estão esperando para serem servidas, ou se muitos objetos estão sendo acessados na mesma requisição através de *scope* [4]. A interação destes componentes é apresentada na Figura 4.5.

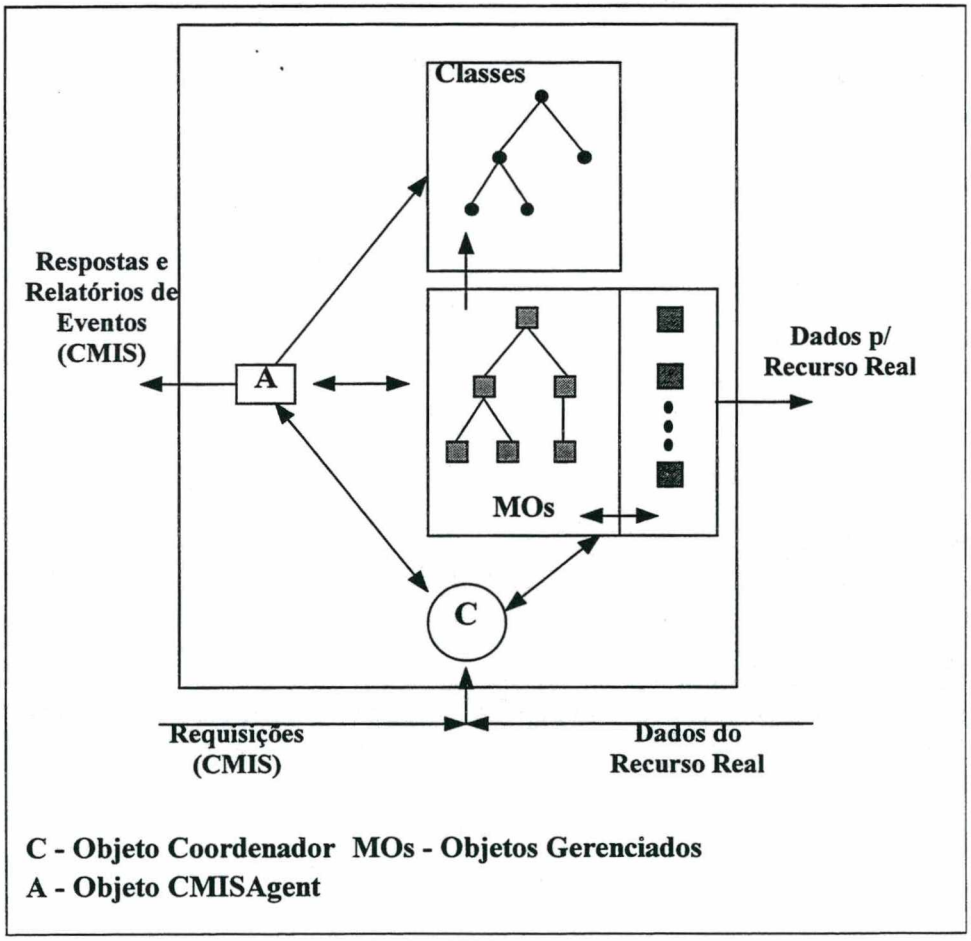

Figura 4.5 - Arquitetura Orientada a Objetos do GMS.

A limitação descrita anteriormente, é contornada pelo Solstice TMN ATK através do suporte que este oferece para a comunicação assíncrona com os MOs. Um estudo para a implementação de uma interface assíncrona entre o CMISAgente e MOs está sendo realizado.

4.5 - Requisitos de Concorrência nos Processos Agentes

Processamento concorrente é fundamental para computação distribuída, podendo ocorrer de muitas formas [33].

Concorrência pode ser aplicada, tanto para as aplicações clientes quanto para servidores. Este estudo, aborda concorrência nos processos servidores, visto que esta é a natureza dos processos agentes (servidores de informação de gerenciamento).

Ao contrário de concorrência no processo cliente, concorrência no servidor requer considerável esforço. Para entender a importância de concorrência nos processos servidores, considere o seguinte exemplo:

Um servidor de acesso remoto, se organizado de forma não concorrente, pode servir somente a uma requisição de cada vez. Outras conexões requisitadas, serão rejeitadas pelo servidor, até a primeira ser finalizada. Isto limita a utilidade do servidor.

Deve ficar claro, que o termo concorrência no servidor, refere-se a possibilidade do processo servidor em manipular requisições concorrentes, não exigindo a implementação de vários processos ou fluxos de execução para alcançar tal funcionalidade.

Em geral, o projeto e implementação de concorrência nos servidores é mais difícil, resultando em um código mais complexo. A escolha de se implementar servidores concorrentes, está na baixa performance e no tempo de espera que servidores não concorrentes causam a seus clientes [33].

4.5.1 - Implementando Concorrência

A implementação de concorrência pode ser realizada das seguintes formas:

(1) Único processo com único fluxo de controle (*Single-Thread*)

(2) Único processo com vários fluxos de controle (*Multithread*)

(3) Vários processos *Single-Thread*

(4) Vários processos *Multithreaded*

Os casos (3) e (4) não são muito utilizados, visto que a criação de vários processos, sejam eles *Single-threaded ou Multithreaded*, degradam consideravelmente os recursos do sistema (Ex. memória, CPU, etc.), desencorajando sua utilização [33].

No caso (1), tem-se a chamada concorrência aparente. Esta é conseguida, utilizando recursos de I/O assíncrono e gerenciamento de estados. Para cada requisição, estados são mantidos, a fim de serem controlados pelo único processo. Este modelo é o adotado para os núcleos apresentados no estudo de casos da seção 4.4.2.

A noção de estados é de difícil implementação, originando um código bastante complexo.

Existem algumas limitações com este tipo de implementação, visto ser um único fluxo de execução dentro de um único processo. Situações de espera em qualquer ponto do código, causa o bloqueio de todo processo. Este modelo pode ser adotado, quando a carga total das requisições submetidas ao servidor, não exceda sua capacidade de processá-las, e que o fluxo de execução dentro do processo garanta a não existência de condições de espera.

O caso (2), sendo atualmente bastante explorado em processos servidores (Ex. de arquivos, de comunicação, etc.), implementa o conceito de concorrência real.

A principal justificativa em se introduzir concorrência real em um servidor, é fornecer rápidos tempos de respostas para múltiplos clientes. Estendendo esta visão para aplicações de gerenciamento atuando no papel de agente, pode se identificar outro fator que encoraja ainda mais sua utilização; a característica de se ter computações internas que são independentes e concorrentes por natureza.

O exemplo mais claro são as execuções dos comportamentos dos objetos gerenciados, juntamente com suas interações com os recursos reais que eles representam. No ambiente real, os recursos reais gerenciados existem de forma independente uns dos outros, em se tratando de seus estados de execução. O exemplo abaixo ilustra este cenário.

Em um ambiente de rede local, um *hub* e uma impressora sendo gerenciados pelo mesmo agente, não possuem qualquer relação entre si. Seus comportamentos são

realizados de forma independente. Conseguir representar o comportamento dos recursos do mundo real, para o ambiente de gerenciamento, é a função dos objetos gerenciados. Portanto, organizar os processos agentes seguindo este modelo, permite que os objetos gerenciados sejam independentes e executem seus comportamentos concorrentemente.

Outro fator que encoraja a utilização deste modelo é a escalabiliade. Dependendo da implementação realizada, os processos servidores podem se adaptar automaticamente com os recursos básicos de *hardware/softwares* oferecidos. Adicionando-se mais processadores ao sistema, por exemplo, o servidor pode ter suas computações automaticamente paralelizadas, uma em cada processador.

A escolha em se usar cada um dos modelos citados, dependerá dos requisitos e dos propósitos a que se aplica os processo servidores. De forma genérica, estes requisitos são:

> **Não Concorrentes**: Utilizados quando o tempo de processamento das requisições é curto, e os tempos de respostas satisfaçam aos requisitos dos clientes.
>
> **Concorrentes**: Quando os requisitos de tempo de respostas são pequenos e os tempos de processamento das requisições são grandes ou desconhecidos.
>
> **Concorrência Aparente**: Quando não se tem o suporte (Ex. sistema operacional, linguagens de programação, etc.) para a implementação de concorrência real e a manipulação e o processamento de várias requisições concorrentes é requerido.
>
> **Concorrência Real**: Quando se tem suporte para sua implementação e requisitos de concorrência são exigidos. Quando se deseja explorar a máxima concorrência oferecida pelo sistema (Ex. máquinas multiprocessadas), esta solução é a única aplicada.

O capítulo seguinte, apresentará uma visão detalhada dos conceitos de *multithreading*, iniciados nesta seção. Estes conceitos serão essenciais para o

entendimento do Capítulo 6, o qual apresenta a proposta e implementação de um núcleo *multithreaded* para agentes OSI.

CAPÍTULO V

Multithreading

5.1 - Introdução

A palavra *multithreading* pode ser entendida como "múltiplas *threads* (fluxo de execução) de controle", enquanto *multithreaded*, a capacidade de se ter *multithreading* [34].

Enquanto um processo tradicional UNIX[1] sempre possui uma única *thread* de controle, *multithreading* (MT) separa um processo em várias *threads* de execução, cada qual executando independentemente no mesmo processo.

Os conceitos básicos sobre *threads,* juntamente com a análise de pacotes de *threads* disponíveis para a implementação de aplicações *multithreaded,* serão assunto deste capítulo.

5.2 - Threads

Na maioria dos sistemas operacionais tradicionais, cada processo tem um espaço de endereçamento e uma única *thread* de controle [35].

Threads compartilham as instruções e dados do processo. Uma alteração em um dado compartilhado (por exemplo, variável global), por uma *thread*, pode ser visto por outras *threads* no processo.

Cada *thread* pode realizar chamadas de sistemas e se comunicar com outros processos (IPC), da mesma forma como processos realizam estas atividades.

Associado a *threads*, mecanismos de sincronização com diferentes semânticas são fornecidos (exclusão mútua, variáveis condicionais, semáforos, etc.). Estes mecanismos podem servir para sincronizar *threads* no mesmo processo ou em processos diferentes. Variáveis de sincronização (instância de um mecanismo de

[1] O conceito de processo UNIX foi utilizado, considerando que o ambiente em que os processos agentes serão inicialmente implementados, será um ambiente UNIX.

sincronização) podem ser colocadas em memórias compartilhadas, sendo possível seu acesso entre *threads* de processos diferentes.

Persistência também é conseguida com a inserção destas variáveis em arquivos, as quais existem, mesmo que seus processos criadores deixem de existir.

Em processos que envolvem concorrência, é desejável que se tenha várias *threads* de controle, sendo que algumas podem ser transitórias e outras podendo existir por toda vida do processo. Concorrência pode ser definida quando, no mínimo, duas *threads* estão em progresso ao mesmo tempo.

Outro conceito diretamente relacionado com *threads* é o de paralelismo. Paralelismo existe quando, no mínimo, duas *threads* estão executando simultaneamente.

Em um processo *multithreaded*, executando sobre um único processador, o processador pode trocar seus recursos de execução entre as várias *threads*, resultando em concorrência. No caso deste mesmo processo executando sobre um multiprocessador com memória compartilhada, cada *thread* pode ser executada em processadores separados, ao mesmo tempo, resultando em paralelismo.

Em determinadas situações, múltiplas *threads* de controle compartilhando o espaço de endereçamento de um único processo, propiciam melhor estruturação e eficiência para a aplicação. *Threads* ou às vezes chamadas *lightweigh process* (processos leves), são ilustradas na Figura 5.1.

Nesta Figura, a máquina (a) possui dois processadores (P1, P2), sendo que P1 está sendo compartilhado por dois processos e P2 dedicado a um terceiro processo. Os processos na máquina (a) possuem uma única *thread* de controle (enfoque tradicional). Na máquina (b), temos um único processo com três *threads* de controle, sendo que duas *threads* estão compartilhando o processador P1 e a terceira *thread*, dentro do mesmo processo, executando em paralelo no processador P2. Na seção 5.5, um ambiente mais complexo reunindo processos, *threads,* processadores e processadores virtuais será apresentado.

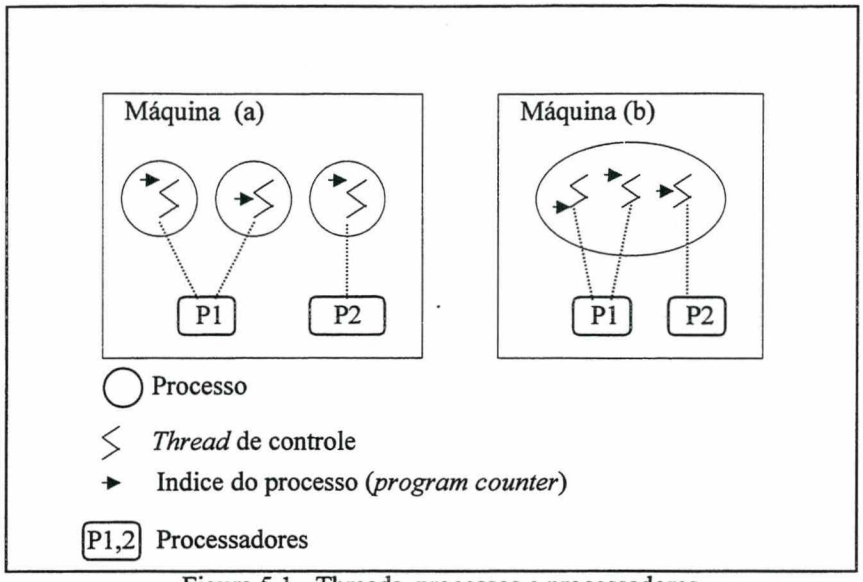

Figura 5.1 - Threads, processos e processadores.

A execução de cada *thread* no processo, se dá seqüencialmente, pois cada *thread* possui seu próprio *program counter* (PC) e pilha. Os Estados de uma *thread* são os mesmos de processos (*Running, Blocked, Ready, Terminate*) [35].

Threads foram criadas a fim de permitir que o paralelismo seja combinado com a execução seqüencial e chamadas de sistema (*system call*) bloqueantes, o que proporciona um alto grau de desempenho em aplicações com tais características (Ex. servidores de RPC).

Vários sistemas adotam uma arquitetura *multithreaded*, encorajados principalmente, pelo alto grau de paralelismo alcançado em máquinas multiprocessadas e por terem inúmeros requisitos de concorrência, o que permitem estruturar suas aplicações em várias computações independentes.

Sistemas servidores de arquivos, servidores de informação para gerenciamento de redes (*management agents*) e etc., são alguns exemplos de aplicações que possuem requisitos de concorrência, as quais são melhor estruturadas utilizando uma abordagem *multithreading*.

Estruturar estas aplicações como uma máquina de estados finito permite simular o paralelismo alcançado com *threads*; contudo, a grande complexidade em se utilizar esta abordagem é um fator desencorajador para sua implementação, visto que estados das diversas computações devem ser alternadamente salvos e restaurados. Chamadas de sistemas bloqueantes não são permitidas, o que aumenta

consideravelmente a complexidade da aplicação, uma vez que chamadas de sistemas não bloqueantes são de difícil programação [35].

Threads são manipuladas por programas de aplicação através de chamadas de sistema. Estas primitivas são organizadas em bibliotecas, comumente chamadas de "pacotes de *threads*" (*threads packages*). A implementação de pacotes de *threads* pode ser feita de duas formas básicas:

- no espaço do usuário;
- no núcleo do sistema operacional (*kernel*).

Na primeira implementação, o *kernel* não sabe da existência de múltiplas *threads* de controle no processo, o qual somente é visto com uma única *thread*. O gerenciamento das *threads* fica a cargo do pacote que está ligado (*linked*) ao código do processo (*Runtime Package*).

Pacotes ao nível de usuário trazem a vantagem de serem passíveis de implementação sobre a maioria dos sistemas operacionais utilizados, permitindo uma grande portabilidade do código que os utilizam (Aplicações).

Em sistemas operacionais que implementam pacotes de *threads* no *kernel*, além das aplicações, o próprio *kernel* se utiliza de *threads* para execução de determinadas tarefas (por exemplo operações de I/O.).

As duas formas de se implementar pacotes de *threads*, possuem vantagens e desvantagens, as quais serão abordadas a seguir.

5.3 - Pacotes de *Threads*: *Kernel* x Espaço do Usuário

Pacotes de *threads* implementados no espaço do usuário, permitem que cada processo tenha seu próprio algoritmo de escalonamento, o que é desejável em alguns tipos de aplicações, como por exemplo, sistemas de tempo real.

O processo de chaveamento entre *threads* do usuário é mais rápido que nas implementações no *kernel*, visto que não existem as perdas decorrentes da comutação do modo usuário para modo *kernel*, a fim de alterar as estruturas (tabelas, filas, etc.) de informações reservadas para as *threads*. O espaço para manter estas estruturas é outra desvantagem nas implementações no *kernel*, visto que a tendência atual em

sistemas operacionais é a implementação de núcleos pequenos (*microkernel*) para fins de distribuição e performance de suas funções.

Threads implementadas no *kernel*, possuem vantagens em relação a *threads* do usuário. Uma das principais vantagens é a possibilidade de se bloquearem em chamadas de sistema (I/O bloqueante, etc.), o que não acontece com a maioria dos pacotes de *threads* implementados no espaço do usuário, onde uma *thread* que se bloqueia causa o bloqueio de todo processo.

Threads no kernel podem executar paralelamente em máquinas multiprocessadas, sendo que este processo de paralelização das *threads*, entre os vários processadores, é transparente para o implementador. A Figura 5.2 apresenta os dois modelos.

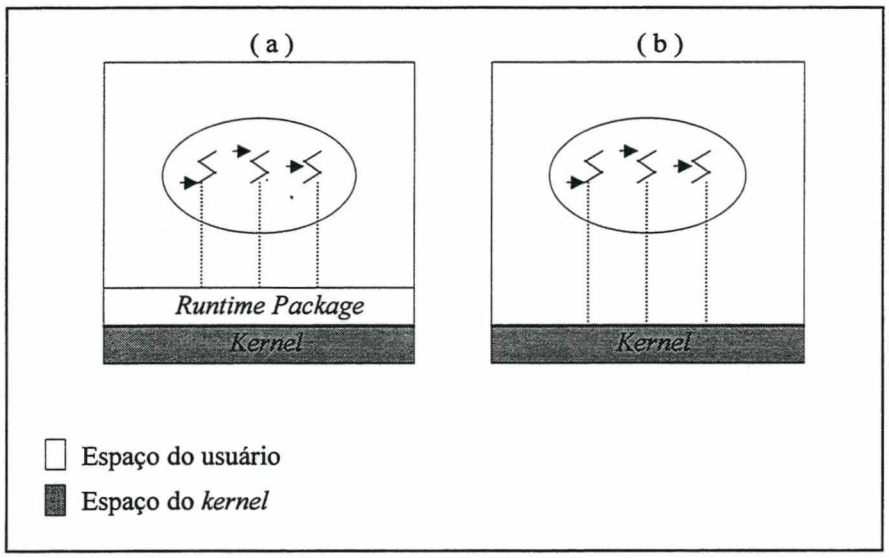

Figura 5.2 - Pacotes de threads no espaço do usuário (a) e kernel (b).

O problema de performance na execução de chamadas de sistema para criação e deleção de *threads* no *kernel*, pode ser resolvido com a reciclagem de *threads* [35].

Quando *threads* são destruídas, estas são marcadas como deletadas e suas estruturas no *kernel* não são desalocadas. Quando uma nova *thread* é criada, a *thread* marcada é reativada, o que incrementa a performance do gerenciamento ao nível de *kernel,* visto que não é necessária a alocação de novas estruturas para a *thread* criada.

Como para o problema anterior, todos os outros problemas relacionados com ambas implementações (*kernel* ou espaço do usuário), possuem soluções alternativas.

Uma destas soluções está relacionada com o problema de bloqueio em chamadas de sistemas realizadas por *threads*, gerenciadas ao nível do usuário. Este problema é resolvido rescrevendo as diversas rotinas do sistema (*read*, *write*, *open*, *getc*, etc.) as quais são implementadas de forma a não se bloquearem. Estas novas implementações normalmente usam chamadas do sistema como *SELECT* [36] para executarem I/O não bloqueante.

Outras implementações usam manipular o sinal de *clock* (*SIGALRM*) para proporcionar um *timeout* no tempo de execução da *thread* que, se estiver bloqueada, não afetará a execução de outras *threads*.

Reescrever rotinas do sistema é um método muito utilizado para garantir a compatibilidade das chamadas de sistema na presença de múltiplas *threads*. Esta característica é denominada *Thread Safe* (TS).

Bibliotecas (stdio, stdlib, etc.) que são TS, garantem que na presença de múltiplas *threads* de controle, seu comportamento é o mesmo do que aquele apresentado com uma única *thread* no processo.

Nesta seção, *threads* foram abordadas de uma forma conceitual. A seguir serão apresentadas algumas implementações reais de pacotes de *threads*.

5.4 - Implementações de Pacotes de *Threads*

Vários sistemas operacionais já trazem suporte para múltiplas *threads* ao nível de *kernel*. Para aqueles sistemas que não incorporam *threads* no *kernel*, existem inúmeros pacotes de *threads* implementados para o espaço do usuário. Na Figura 5.3 são apresentados alguns exemplos de sistemas operacionais e seu suporte para *multithreading*.

Sistema Operacional	**Pacote de *Threads***
SunOs 4.0™	Espaço do usuário
SunOs 5.x (Solaris 2.x)	*Kernel*
AMOEBA	*Kernel*
MACH	*Kernel*
CHORUS	*Kernel*
SCO	Espaço do usuário

Linux	Espaço do usuário
Windows NT©	*Kernel*
OS/2™	*Kernel*
AIX 4.1	*Kernel*

Figura 5.3 - Exemplos de suporte a multithreading para alguns sistemas operacionais.

Vale lembrar que sistemas que suportam *threads* no *kernel*, também podem ter *threads* no espaço do usuário, sendo estas disponibilizadas através de bibliotecas. Um exemplo é o MACH, que além de suportar *threads* no *kernel*, possui a biblioteca *C-threads*, implementada no espaço do usuário.

Uma terceira abordagem para pacotes de *threads*, são aqueles implementados no espaço do usuário e que trabalham em conjunto com *threads* do *kernel*. Fica claro que para ter este tipo de implementação, é necessário que o pacote de *threads* tenha sido escrito para aproveitar os recursos do sistema operacional, que suporta *threads* no *kernel*. Um exemplo desta implementação será visto na apresentação da biblioteca *libthread* do Solaris 2.

Dentre os exemplos acima, dois serão abordados com mais detalhes, sendo uma implementação no espaço do usuário e outra no *kernel*. A implementação do pacote de *threads* ao nível de *kernel,* será a do Solaris 2, visto ser o sistema operacional utilizado na implementação da plataforma. Outro pacote analisado será o *pthreads*, implementado no espaço do usuário, e amplamente utilizado em vários sistemas. Uma característica que levou a escolha deste pacote, é que sua implementação é baseada no padrão POSIX P1003.4a [37], o que permite total portabilidade para as aplicações que o utilizam.

A tendência atual, é que tanto *threads* no espaço do usuário quanto no *kernel*, sigam o padrão POSIX P1003.4a.

5.5 - *Threads* no Solaris 2.x

Na versão 4.0 do SunOS, o suporte a *multithreading* era realizado pela biblioteca LWP (LWP *library*), a qual é um pacote de *threads* para o espaço do usuário. Não existia suporte para *threads* no *kernel*. Com o Solaris 2.0, o SunOS

passou a suportar *threads* ao nível de *kernel,* sendo sua arquitetura *multithreaded* apresentada na Figura 5.4.

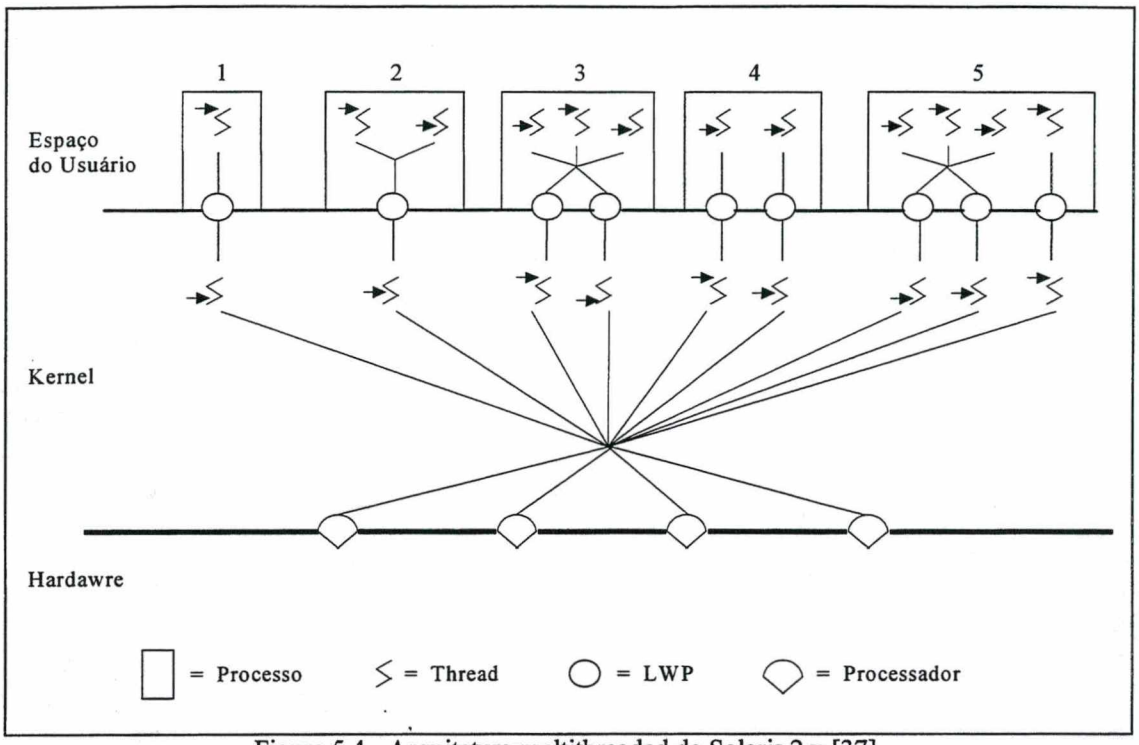

Figura 5.4 - Arquitetura multithreaded do Solaris 2.x [37].

Além de *threads,* processos e processadores, o conceito de LWP[1] foi utilizado. LWPs na arquitetura *multithreaded* do Solaris 2.x, é o termo utilizado para representar um conceito de CPU virtual, a qual executa o(s) código(s) da(s) *thread*(s) do processo e chamadas de sistema para o processo. LWPs é a ponte entre o nível do usuário e o nível do kernel. Cada processo contém uma ou mais LWPs, cada qual executando uma ou mais *threads* do processo.

Cada LWP possui associada uma *thread* no *kernel*, que pode estar executando sobre um dentre os vários processadores, caso existam. Portanto, *threads* são implementadas utilizando LWPs e são atualmente representadas por estruturas de dados no espaço de endereçamento do processo, ao contrário de LWPs que são representadas por estruturas no *kernel*.

Programadores normalmente utilizam *threads* e não LWPs. A interação com as LWPs é função da biblioteca de suporte a *multithreading* (*libthread*).

[1] LWPs na arquitetura do Solaris 2.x, não tem nenhum relacionamento com a biblioteca LWP do SunOS 4.0. Esta biblioteca não é suportada pelo Solaris 2.x, o qual utiliza uma outra biblioteca chamada *libthread.* O mesmo nome (LWP) para conceitos diferentes é uma simples coincidência [34].

Muitos programadores utilizam *threads*, no Solaris 2, sem saber da existência de LWPs, não sendo muito comum processos do usuário interagirem diretamente com LWPs [37]. Em alguns casos particulares, esta interação é necessária, visto que os programadores podem implementar noções particulares de concorrência manipulando LWPs diretamente.

Algumas aplicações requerem uma grande quantidade de paralelismo lógico (*threads*), enquanto outras necessitam de um paralelismo real (LWP), a fim de mapear suas computações sobre os vários processadores disponíveis no sistema. Em ambos os casos, programas podem ter acesso aos dois serviços.

Sendo implementadas por biblioteca e executando no espaço do usuário, *threads* são gerenciadas (criadas, destruídas, bloqueadas, etc.) sem envolver o *kernel*. Já o gerenciamento de LWPs é realizado pelo *kernel*, e comparado com *threads*, é bem menos eficiente. Processos com múltiplas *threads*, que estão mapeadas cada uma sobre LWPs, podem ser pouco eficientes, dependendo do número de LWPs requeridas. Como pode ser observado na Figura 5.4, nos processos (1 a 5), várias configurações são possíveis, dependendo dos requerimentos do usuário.

A biblioteca de *threads* permite que o usuário crie *threads* permanentemente associadas a LWPs (*Bound threads*) ou *threads* escalonadas sobre várias LWPs (*Unbound threads*). *Bound threads* são utilizadas para se ter *threads* escalonadas globalmente, o que é muito útil em aplicações com requisitos de tempo real.

Unbound threads são mais leves e podem executar sobre as várias LWPs disponíveis, não sendo escalonadas globalmente, visto não possuírem associadas a elas uma LWP do sistema. A escolha em se utilizar *bound* ou *unbound threads* depende de seu papel no processo, sendo uma decisão do programador da aplicação.

A Tabela 5.1 apresenta alguns resultados de performance em operações com *threads*, *LWPs* e processos.

	Tempos em (μ segundos)	
	Criação	Sincronização com semáforo
Thread (espaço do usuário)	52	66
LWP	350	390
Processo	1700	200

Tabela 5.1 - Latência de *Threads* do usuário, LWP e processos sobre uma SPARCstation 2.

5.6 - Pthreads

O grupo conhecido como POSIX 1003.4a tem trabalhado sobre um padrão para programação *multithreaded*. A interface de *threads* do Solaris 2, citada anteriormente, difere da interface POSIX, contudo, não existe diferença nas funcionalidades oferecidas pelas duas interfaces. Contudo, a partir do Solaris 2.5, além do pacote de *threads* apresentado anteriormente, este traz uma implementação do *pthreads*.

Atualmente, *pthreads* está sendo implementada como biblioteca no espaço do usuário, sendo utilizada sobre várias plataformas de hardware (SPARCstations, DECstation™ r2000/r3000, Intel, etc.) e sistemas operacionais (HP-UX, SunOS 4/5, Linux, SCO, etc.).

Na implementação do núcleo *multithreaded*, toda manipulação de *threads* segue o padrão POSIX 1003.4a. Foi utilizado como pacote *pthreads*, a implementação realizada por Chris Provenzano (proven@mit.edu), a qual tem como característica principal, ser portável dentre as plataformas de hardware e sistemas operacionais mais utilizadas atualmente. Sendo assim, para aqueles ambientes que não oferecem nativamente uma biblioteca *pthreads*, pode-se utilizar esta implementação.

Para aquelas plataformas que já suportam *pthreads*, como é o caso do Solaris 2.5 nenhuma alteração no código é efetuada, tendo a total portabilidade garantida.

Para a realização da implementação e testes, o pacote *pthreads* foi instalado em um PC/i486 com sistema operacional Linux 1.2.8.

Além da implementação do núcleo, alguns testes com este pacote foram realizados, e uma característica interessante é que mesmo sendo implementado no espaço do usuário, uma *thread* que se bloqueia em uma chamada de sistemas, não interfere na execução de outras atividades do processo. Este foi outro fator relevante para sua escolha.

O pacote *pthreads* escolhido, também utiliza o conceito de bibliotecas TS, o qual assegura que chamadas de sistema (por exemplo, *malloc*) se comportarão na presença de múltiplas *threads* como se estivessem sendo executadas em um processo com uma única *thread* de controle.

Neste capítulo foram apresentados conceitos de *multithreading*, exemplos de implementações reais de pacotes de *threads*, e algumas justificativas para a escolha do pacote *pthreads* adotado na implementação do núcleo. A seguir serão apresentados a proposta e implementação do núcleo *multithreaded* para agentes de gerenciamento.

CAPÍTULO VI

Núcleo *Multithreaded* para Agentes de Gerenciamento OSI

6.1 - Introdução

Os objetivos do gerenciamento de redes são imutáveis, contudo, o processo pelo qual estes objetivos são alcançados, tem-se alterado de maneira a fazer uso dos atuais avanços tecnológicos.

Com novas tecnologias e serviços sendo introduzidos a cada dia, sistemas de gerenciamento de redes devem estar prontos para atender tais inovações, utilizando-se dos atuais avanços tecnológicos e oferecendo cada vez mais funcionalidades que atendam aos atuais requisitos de gerenciamento [38] [39].

Tecnologias para controle automático de decisões, correlação de alarmes, arquiteturas seguras e de alta performance são alguns exemplos de novas evoluções que devem ser aplicadas aos sistemas de gerenciamento de redes, a fim de torná-los eficientes e capazes de atender os atuais requisitos de gerenciamento [40].

Atualmente, devido ao avanço dos sistemas operacionais, linguagens de programação e ambientes de suporte ao desenvolvimento de aplicações, a implementação de aplicações *multithreaded* tem se tornado freqüente em inúmeras áreas:

- Sistemas distribuídos (servidores de arquivos, servidores de RPCs, etc.);
- *Browsers* WWW (HotJava, etc.);
- Ambientes para programação paralela (TPVM, LPVM, etc.);
- Sistemas de Banco de Dados (SQLServer, etc.), dentre outros.

Frente a estas evoluções, este capítulo apresenta a proposta e implementação de um núcleo para agentes de gerenciamento, o qual se utiliza de recursos de *multithreading* com o objetivo de oferecer maiores funcionalidades e performance às aplicações agentes da plataforma em desenvolvimento.

6.2 - Modelo

Como apresentado na seção 4.3, processos agentes devem suportar características de interação entre os objetos gerenciados e os recursos que eles representam.

Atendendo aos requisitos definidos no capítulo 4, os serviços do núcleo abordam os seguintes aspectos:

- Aspectos de tratamento das operações de gerenciamento e emissão de notificações;
- Dinâmica (comportamento) dos objetos gerenciados.

Com relação ao tratamento das operações de gerenciamento e emissão de notificações, foi definida uma política de serialização destas operações. O atendimento das operações, requisitadas pelo gerente, é feito de maneira serializada, de acordo com a ordem de chegada destas operações (*First-Come-First-Served*).

A serialização das operações de gerenciamento se dá no momento em que estas são enviadas ao processo agente, pelo ACSE/CMIP. Neste ponto, a organização da estrutura de entrada do núcleo, se encarrega de agrupar as operações que chegam, seguindo uma política FIFO.

Outras formas de se estruturar processos servidores, relacionadas com invocações externas de operações foram estudadas, sendo que uma das mais utilizadas para aumentar a performance destes processos, é semelhante à implementação de um servidor de RPC *multithreading* [35]. Para cada operação que chega, uma *thread* é criada para a execução desta operação.

Tal organização não pode ser adotada, visto que a serialização das operações não é garantida. Para os casos de operações realizadas sobre o mesmo objeto, o protocolo CMIP requer a serialização destas operações, ao contrário de operações não sincronizadas sobre objetos diferentes, que podem ser executadas sem nenhum requisito de serialização, referente à ordem de emissão das operações pelo gerente.

Um exemplo que invalida a utilização do enfoque adotado para o servidor de RPC *multithreading* poderia ser a execução de operações, não serializadas, sobre o mesmo objeto. Um gerente solicitando uma operação SET sobre atributos de vários

objetos, utilizando escopo e filtro, em seguida emite uma operação GET para recuperar o valor de um atributo alterado, em um dos objetos afetados na operação anterior. A operação SET poderia perder o processador antes de ser concluída, o GET tomando o processador, poderia retornar o valor do atributo, o qual não foi alterado pela operação SET, criando uma situação de erro.

Em função das exigências de serializações do protocolo CMIP, foi considerado que operações de gerenciamento serão tratadas seqüencialmente. Um objeto de suporte OM (*Object Manager*) foi criado, o qual tem a função de receber as operações de gerenciamento e repassá-las aos MOs (*Managed Objects* - Objetos Gerenciados). Tanto o OM quanto os MOs possuem filas de operações seguindo uma política FIFO.

Para cada requisição, o OM verifica se esta deve ser tratada por ele, ou simplesmente repassá-la aos objetos gerenciados.

Após chegar mensagens em sua estrutura de entrada, o OM não necessita ser invocado, pois ele possui sua própria *thread* de controle, e constantemente estará consumindo mensagens de sua fila, caso existam.

Quando não existem mensagens para serem consumidas, o OM continua executando partes de seu comportamento (mudança de prioridade entre os objetos, sincronização de operações, etc.) relativas ao gerenciamento dos vários objetos gerenciados da MIB.

Cada objeto, possuindo sua própria fila de operações, possibilita que sua execução seja realizada de forma independente e concorrente com outras atividades do agente, incrementando a sua performance. A Figura 6.1 apresenta a infra-estrutura proposta.

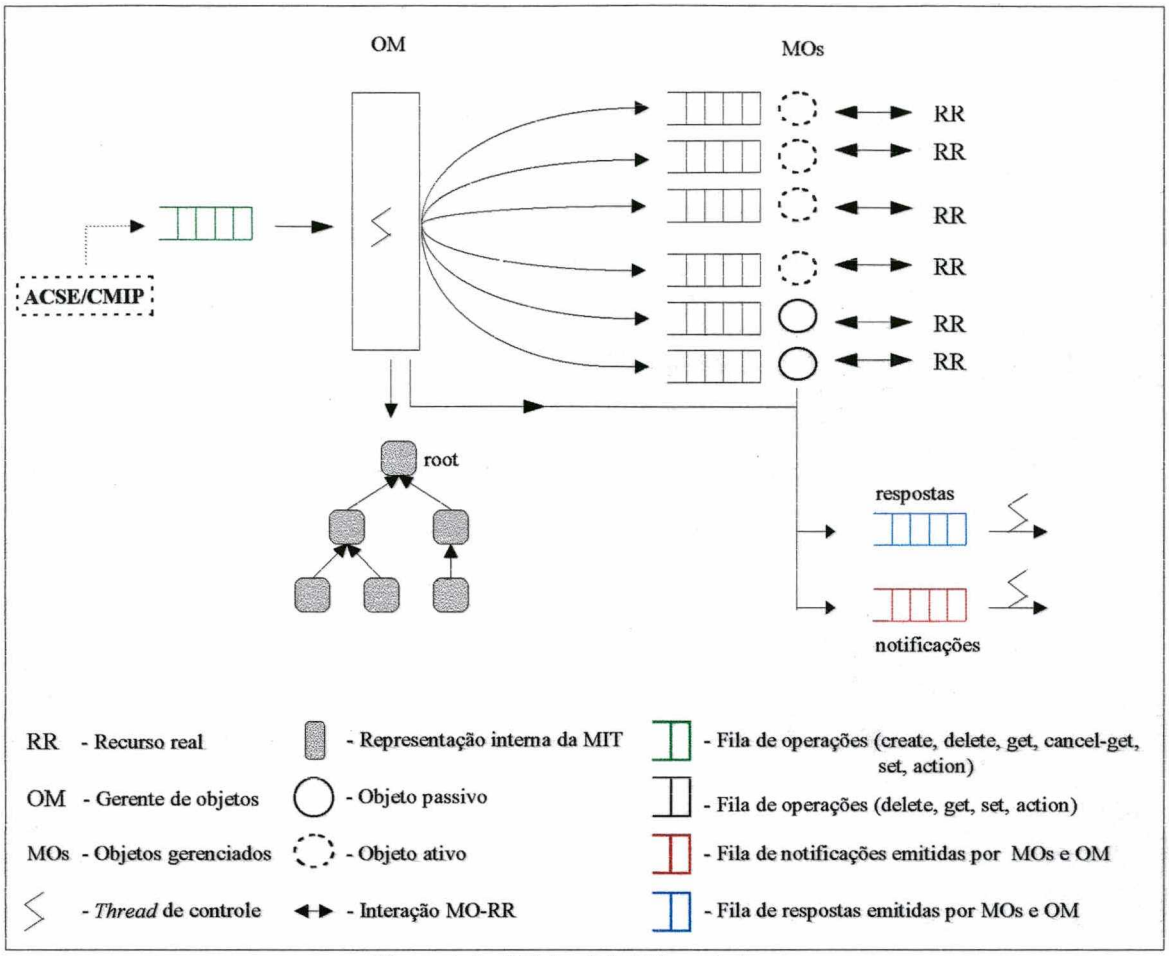

Figura 6.1 - Núcleo Multithreaded.

Para o propósito de independência e concorrência entre os objetos gerenciados, o modelo adota o conceito de objetos ativos, na implementação dos objetos gerenciados.

6.3 - Objetos Ativos

No paradigma orientado a objetos (OOP), relacionado aos estados de execução dos objetos, podemos classificá-los em dois tipos básicos: **ativos** e **passivos**.

Segundo Booch [41], concorrência em OOP é a propriedade que distingue um objeto ativo de um objeto não ativo (passivo).

Na presença de concorrência, é apropriado que se visualize os objetos como entidades ativas autônomas, as quais executam seu comportamento independente de invocações externas [42] [43].

Utilizando os conceitos apresentados até o momento, podemos dizer que objetos ativos são aqueles que encapsulam sua própria *thread* de controle, a fim de

possuírem autonomia para mudarem seu estado interno, sem a necessidade de invocações externas.

Objetos passivos não encapsulam sua própria *thread* de controle, e só podem mudar seu estado na presença de uma invocação externa, que pode ser tanto por outro objeto ativo ou pela *thread* de controle do processo em que o objeto faz parte.

Após ser invocado, o objeto passivo torna-se ativo, realiza alguma computação (seu comportamento) e retorna ao estado passivo.

Algumas linguagens [44][45] orientadas a objeto fornecem suporte à implementação de objetos ativos, contudo, a linguagem C++, a qual está sendo utilizada na implementação da plataforma, somente implementa o conceito de objetos passivos.

Em função desta limitação, foram utilizadas *threads* associada com a linguagem C++, a fim de resolver esta restrição e permitir que o conceito de objetos ativos seja então implementado.

6.4 - Objetos Gerenciados como Objetos Ativos

Com o objetivo de melhor atender aos requisitos de gerenciamento, adotou-se a utilização de objetos ativos para implementar objetos gerenciados, visto que a representação de recursos reais através de objetos ativos, se encaixa melhor com a abstração de recursos do sistema (entidade de camada, conexão de transporte, etc.) [46].

Enfoques tradicionais (objetos passivos) limitam a implementação de tais abstrações, devido a falta de dinâmica independente entre os objetos. Um problema com estas limitações pode ser verificado no exemplo a seguir.

Um agente com dez objetos passivos, em um dado instante, somente poderá estar executando o comportamento de um único objeto (enfoque tradicional). O objeto em execução, bloqueia-se em um evento qualquer (computação de uma mensagem, interação com um recurso, *bug* de *software*, etc.), agora nenhum objeto pode se tornar ativo, nem mesmo o processo volta a executar, pois sua única *thread* de controle, está em posse do objeto bloqueado.

Para resolver este problema, a maioria dos sistemas restringem a implementação dos objetos, pois não são permitidos bloqueios em eventos externos ou até mesmo executando uma computação mais complexa.

Um exemplo de tais sistemas, é a plataforma de gerência OSIMIS, a qual define que implementações de objetos gerenciados não podem se bloquear em pontos externos de comunicação, causando o bloqueio de todo processo agente [17].

Restrições, como esta, limitam a implementação dos objetos gerenciados, no que se refere a interações com recursos do sistema e computações que exijam maior complexidade.

Em casos de interações que dependam de comunicações proprietárias à interface do recurso, estas não podem ser fornecidas pela plataforma, tendo que ser implementadas pelo próprio comportamento do objeto, o que também não seria possível, frente as limitações impostas pela plataforma, com relação a expor seus objetos em interações com recursos que poderiam causar um bloqueio do objeto e, consequentemente, de toda aplicação.

Se no exemplo anterior, o modelo fosse de objetos ativos, o objeto que se bloqueou não afetaria outros objetos, nem o próprio processo, visto que suas *threads* individuais estariam livres para executarem suas atividades.

Objetos ativos refletem com maior fidelidade a principal função dos objetos gerenciados; representar recursos reais.

O núcleo *multithreaded* (Figura 6.1) não impede a utilização de objetos passivos, pois em alguns casos, objetos gerenciados possuem um comportamento estático e serão melhor implementados como objetos passivos.

Um exemplo para objetos gerenciados implementados como entidades não ativas, são objetos da classe *log*, os quais somente realizam suas funções mediante invocações externas (gravação, leitura, etc.). Nestes casos, é desejável que o objeto seja implementado como passivo, pois sua atividade constante, somente iria consumir recursos de processamento (CPU) sem necessidade.

Como dito anteriormente, objetos passivos podem se bloquear e comprometer a operação de outros objetos passivos ou da *thread* de controle que o invocou. Para isto, foi proposta uma implementação de objetos passivos, a qual

permite que um eventual bloqueio em função de seu comportamento, não comprometa a execução de outras atividades no processo.

Esta implementação foi possível, dadas as características *multithreaded* do núcleo, e será abordada na seção 6.7.

6.5 - Dinâmica dos Objetos

A execução concorrente dos objetos proporciona, ao sistema gerente, menores tempos de resposta às operações requisitadas, e uma visão mais precisa da MIB, no que se refere aos estados dos objetos gerenciados e os estados dos recursos que estes representam.

Como citado no capítulo 4, objetos podem obter informações de seus recursos acessando-os de três formas básicas:

- mediante requisição do gerente;
- através de *polling* periódicos;
- recebendo alarmes (*traps*) de recursos com funções de gerência embutidas.

No primeiro caso, organizações tradicionais[1] causariam o bloqueio temporário de toda aplicação, em função do objeto estar interagindo com um recurso naquele dado instante. Caso este recurso estivesse fracamente acoplado ao sistema, várias requisições posteriores poderiam ficar pendentes por um longo período, esperando o término da comunicação entre objeto e o recurso. Utilizando a infra-estrutura proposta pelo núcleo *multithreaded*, todas as atividades do agente serão independentes, proporcionando maior justiça em suas execuções e não permitindo que computações ativas em um dado instante, monopolizem a utilização do processador. A possibilidade destas computações não devolverem o processador, em função de uma falta qualquer (quebra em sua execução normal), não ocorre com o núcleo proposto.

No segundo caso de interação, tanto organizações tradicionais quanto a organização proposta pelo núcleo *multithreaded*, fornecem um suporte desejável para

[1] Processos agentes que tem a interface com os MOs realizada de forma síncrona e são *Single-Threaded* (Ex. OSIMIS)

sua implementação. Contudo, no enfoque *multithreaded*, tem-se a facilidade de fornecer a determinados objetos gerenciados, maiores prioridades com relação às suas execuções, causando uma maior freqüência de suas operações de *polling*. Isto é desejável pois, em alguns recursos, atualizações mais precisas são necessárias, a fim de suportarem características de tempo real. Em aplicações agentes, atuando no gerenciamento de faltas, esta facilidade é desejável no sentido de proporcionar uma maior precisão na detecção de problemas que venham a ocorrer com determinados recursos que são vitais para o perfeito funcionamento do sistema.

A terceira interação é suportada em ambas organizações. Semelhante ao caso anterior, determinados recursos que exigem uma atenção especial (Ex. *aplicações/dispositivos* de segurança), podem ser privilegiados em ter seus alarmes atendidos de forma prioritária a outros objetos gerenciados que exigem menor atenção.

A função de alterar prioridades entre os objetos gerenciados é delegada ao OM, o qual interfere diretamente nas prioridades do conjunto de *threads* utilizadas pelos objetos gerenciados.

O projetista da aplicação agente será encarregado de definir quais prioridades serão atribuídas para cada objeto.

Com o objetivo de incrementar ainda mais a performance do processo agente, objetos podem passar a sua vez de execução para outras entidades ativas do sistema, verificando que, em um dado instante, seu comportamento não realizará nenhuma computação.

Filas vazias, atributos de estado operacional desabilitados (*disable*), recursos gerenciados temporariamente indisponíveis, são alguns exemplos de condições para que os objetos repassem o processador para outras atividades, a fim de não consumirem toda sua fatia de tempo de execução sem nenhuma computação.

Objetos passivos somente realizam operações mediante invocação do OM, ao contrário dos objetos ativos que estão constantemente verificando suas filas de mensagens.

A serialização das operações se dá ao nível de operações sobre o mesmo objeto, sendo que, operações que exigem sincronização e operações sobre objetos que se relacionam com outros objetos, devem ser tratadas em especial. O primeiro caso

será tratado na seção 6.6, pois este exige uma descrição mais detalhada. No segundo caso, objetos mantêm relacionamentos com outros objetos através de atributos de representação de relacionamentos (ARR) [8]. Estes relacionamentos serão modelados de acordo com as regras definidas em [47].

Requisições que utilizam escopo, filtro e sincronização, devem previamente serem tratadas pelo OM, pois além de gerenciar o acesso a MIT (*scope*, *filter*), este fornece mecanismos para a implementação de sincronizações entre objetos, dentre outras funcionalidades.

No modelo apresentado pela Figura 6.1, existem explicitamente três *threads* principais. Uma é dedicada à execução do OM, as outras duas são para cada uma das filas de respostas e notificações, as quais são dedicadas à emissão destas mensagens. É função do OM, realizar a manutenção (criar, deletar, suspender, reativar, alterar prioridades, etc.) destas *threads* e das *threads* de cada objeto gerenciado, ao nível da aplicação agente.

Abaixo, estão listadas as principais tarefas que o OM realiza para cada uma das operações de gerenciamento:

OPERAÇÃO CREATE

- Cria uma fila de operações para o objeto;
- Cria uma instância da classe requisitada na operação;
- Cria uma *thread* de controle para o objeto;
- Associa ao objeto criado a *thread* de controle;
- Associa ao objeto a fila de operação;
- Atualiza a MIT
- Inicializa os valores dos atributos deste objeto com algum objeto de referência, se especificado no *Name Binding* [6];
- Emite um relatório de eventos notificando a criação e atualiza seu estado interno (tabelas e variáveis do núcleo), a fim de suportar o objeto criado.

A *thread* associada ao novo objeto, executará seu método *behaviour()*, sendo que parte deste comportamento é definido pela plataforma e o restante será aquele definido no construtor (BEHAVIOUR) do *template* da classe de objeto gerenciado.

Parte do comportamento fornecido pela plataforma está relacionada com o tratamento da fila de operações do objeto gerenciado e com interações com objetos de sincronização (*S*), sendo esta parte da implementação denominada "código de suporte aos objetos ativos".

OPERAÇÃO DELETE

- Validação das regras de deleção (caso existam definidas no *Name Binding* da classe do objeto deletado);
- Aplica regras de escopo e filtro, caso requisitadas na operação;
- Deleta a fila de operações do objeto (Se nenhuma operação pendente);
- Deleta o objeto;
- Finaliza a *thread* de controle do objeto (*thread_join*);
- Atualiza a MIT
- Emissão de um relatório notificando a deleção do objeto e atualização de seu estado interno.

OPERAÇÕES GET, SET, ACTION

- Localiza o objeto base na MIT;
- Aplica regras de escopo e filtro, caso requisitadas na operação;
- Recupera os identificadores das filas para cada um dos objetos selecionados (zero ou mais) na operação;
- Insere a operação nas respectivas filas, se um ou mais objetos forem selecionados;

OPERAÇÃO CANCEL-GET

Esta operação exige algumas explicações. Como pode ser observado na Tabela 6.1 [4], esta operação possui somente três parâmetros.

Nome do Parâmetro	Req/Ind	Rsp/Conf
Identificador da Invocação	Obrigatório	Obrigatório
Identificador do GET a ser cancelado	Obrigatório	Não presente
Erros	Não presente	Condicional

Tabela 6.1 - Parâmetros do M-CANCEL-GET.

Esta operação, ao contrário das demais, não refere-se a uma instância de classe de objeto gerenciado, e sim, à operação M-GET que será cancelada. A operação M-GET será cancelada se estiver em trânsito dentro do agente, ou esteja em um dos seguintes estados:

- Pronta para ser executada, mas não executando;
- Executando;
- Finalizada, mas não confirmada (ainda não foi efetuado o envio de sua resposta).

Estando em um destes estados, a operação M-GET pode ser abortada por um M-CANCEL-GET.

Para realizar o M-CANCEL-GET, o OM consulta as filas de operações e respostas (Figura 6.1). Um método do MO também deve ser invocado, pois a operação pode estar no estado executando, não aparecendo em nenhuma das filas citadas.

OPERAÇÃO EVENT-REPORT

Esta operação não envolve o OM, pois sua emissão é diretamente realiza pelo MO. Objetos de suporte para a emissão de relatórios de eventos, são invocados pela *thread* dedicada a emissão de relatórios de eventos (ver Figura 6.1).

Com relação a todos os procedimentos citados para a realização das operações de gerenciamento, vale ressaltar que objetos gerenciados são agora tratados como objetos ativos, o que exige que o comportamento especificado para estas

operações, seja garantido sobre o ambiente *multithreaded*. Assim as implementações seguem um regime *thread safe* (TS).

6.6 - Operações que Exigem Sincronização

Cada objeto tendo sua própria fila de operação e executando de forma independente e concorrente com outros objetos do sistema, cria um problema em situações onde uma operação deve ser executada sobre múltiplos objetos e requer sincronização.

O protocolo CMIP permite duas formas de sincronização:

- atômica (*atomic*);
- melhor esforço (*best effort*).

Na sincronização com melhor esforço, a falha na execução da operação por algum objeto do grupo não interfere na execução dos outros objetos.

A sincronização atômica exige que todos objetos do grupo tenham sucesso na execução da operação; caso algum objeto falhe, todos objetos envolvidos abortam a operação, mesmo sendo capazes de executá-la.

O problema com este tipo de operação, pode ser ilustrado no exemplo a seguir.

Suponha-se que, para uma operação de gerenciamento SET (S), solicitada com escopo e filtro, três objetos foram selecionados, tendo esta operação sido requisitada com sincronização atômica.

Como cada objeto tem sua própria fila de operações, a Figura 6.2 ilustra o cenário do problema.

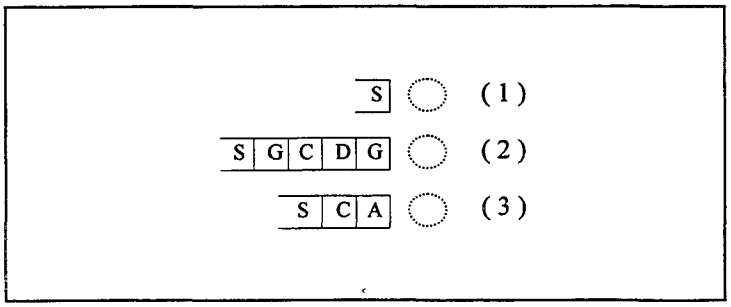

Figura 6.2 - Cenário para o problema de sincronização.

Como o objeto (1) não possui mensagens de operações em sua fila, a operação SET será a única da fila de operações deste objeto. No caso do objeto (2), existem operações pendentes, e o SET será a quinta operação da fila. O terceiro objeto possui somente duas operações em sua fila, e a operação SET será a terceira a ser atendida.

Neste cenário, verifica-se o problema de que a operação (S) poderá ser executada pelos objetos em tempos diferentes. Para resolver o problema, um objeto de suporte à aplicação agente, denominado objeto sincronizador (*S*), coordenará operações que exijam sincronismo atômico. A Figura 6.3 apresenta a solução.

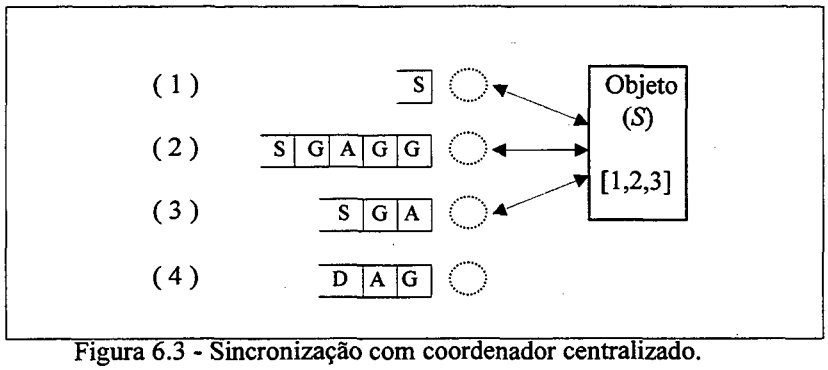

Figura 6.3 - Sincronização com coordenador centralizado.

Quando operações com sincronização (atômica) forem submetidas aos objetos, o OM criará um objeto (*S*), o qual conterá os identificadores dos objetos que estão no grupo da operação. Neste exemplo, somente os objetos 1, 2 e 3 estão sendo submetidos à sincronização atômica, sendo que seus identificadores estão contidos no objeto (*S*).

Quando o objeto gerenciado retirar a próxima operação de sua fila, este verificará que se trata de uma operação que exige sincronismo atômico, o objeto gerenciado notificará ao objeto sincronizador que está pronto para executar a operação. A partir deste momento, o objeto gerenciado espera por uma mensagem do objeto sincronizador, solicitando a sua execução da operação. Esta mensagem, somente será emitida pelo objeto *S* quando todos objetos, pertencentes ao grupo, estiverem prontos para executar a operação. Sendo assim, novas operações que chegam na fila do objeto, não são tratadas, até que ele seja liberado pelo *S*. Outras atividades podem ser executadas pelo objeto gerenciado, a fim de manter sua imagem do recurso real sempre atualizada.

Após o objeto *S*, receber de todos objetos do grupo, notificações de que estão prontos, este envia uma mensagem para os objetos executarem a operação.

Após executarem a operação, os objetos gerenciados devem enviar ao objeto *S* uma resposta de que a operação foi realizada com sucesso ou falha, pois na execução da operação sobre os recursos gerenciados (recursos reais), podem ocorrer problemas na interação (objeto-recurso).

Caso alguma resposta de falha seja notificada, a transação é cancelada para todos objetos do grupo, que após retornarem para o estado anterior, são liberados para continuarem a servir suas filas de operações.

O objeto *S*, resolve o problema da sincronização atômica, contudo, em alguns casos, ele cria um novo problema.

Assumindo o cenário anterior, objetos que estão prontos para executarem a operação, emitem uma notificação para o objeto *S* e ficam aguardando uma mensagem para que possam executar a operação e serem liberados.

Supondo que os objetos 1 e 2 estejam prontos para executarem a operação, e o objeto 3, tratando a operação GET (G), fique bloqueado, indefinidamente, por algum motivo (indisponibilidade de um recurso, *bug* de *software*, etc.), os objetos 1,2 e *S*, ficarão esperando indefinidamente, sendo que suas filas de operações podem estar com várias mensagens pendentes.

Sendo a comunicação entre os objetos gerenciados e o objeto *S* assíncrona e, assumindo que objetos são passíveis de falhas, visto que seu comportamento é uma implementação do usuário, foi identificado o problema de não ser possível um acordo entre os objetos gerenciados e objetos *S*, quando algum objeto gerenciado do grupo venha a falhar.

Fischer et al. (1985) provou que em um ambiente onde entidades se comunicam com tempo de transmissão indefinido, e pelo menos uma destas entidades sendo passível de falha, nenhum acordo é possível [35]. Em função destas características, se o terceiro objeto não responde, nenhuma conclusão sobre o estado do objeto pode ser realizada, visto que ele pode estar realmente com problemas (*crash*) ou pode estar executando uma computação lenta.

A impossibilidade do objeto *S* em distinguir se objetos gerenciados estão em computações lentas ou em *crash*, pode ser amenizada com a solução descrita em [48].

Adaptando esta solução para o cenário anteriormente descrito, o objeto *S* terá a capacidade de suspeitar que um determinado objeto está em *crash*, o que faz com que a operação seja abortada, e todos objetos que estavam esperando sejam liberados.

Fica claro em [48], que suspeitas erradas podem ocorrer e, nestes casos, o objeto não estaria em *crash* e a operação seria abortada, o que exigiria do gerente uma nova retransmissão da operação.

A capacidade do objeto *S*, em realizar suspeitas de falhas nos objetos gerenciados, é conseguida inserindo-se um tempo de resposta (*timeout*) em cada objeto do sistema, sendo que este tempo pode ser definido pelo implementador do objeto, ou assumir um tempo padrão (*default*), especificado pela plataforma.

O tempo especificado pela plataforma busca atender a maioria dos casos, visto que a implementação do objeto é realizada pelo usuário e não se tem como prever seu tempo de resposta.

Problemas e soluções relacionados, podem ser encontrados em [49] [50] [51].

6.7- Objetos Passivos não Bloqueantes

A fim de permitir que a implementação de alguns objetos gerenciados (*log*, registro de *log*, etc.) modelados como objetos passivos, não comprometa outras atividades do sistema, caso estes venham a se bloquear, foi definido que cada objeto passivo possuirá sua própria *thread* de controle.

Estes objetos somente se tornarão ativos na presença de invocações externas, preservando a semântica de objetos passivos.

Nesta implementação, o OM não acessa diretamente o método do objeto, visto que, se o método se bloquear por algum motivo, a *thread* do OM será bloqueada e não poderá atender às novas PDUs que chegam do CMIP ou ACSE.

Portanto, o OM solicita que a *thread* do objeto passivo invoque seus métodos para atender as operações de sua fila, não afetando assim a execução de outras atividades do agente, caso o objeto passivo se bloqueie.

A Figura 6.4 ilustra, conceitualmente, a semântica do comportamento dos objetos passivos e ativos, ambos com suas próprias *threads* de controle.

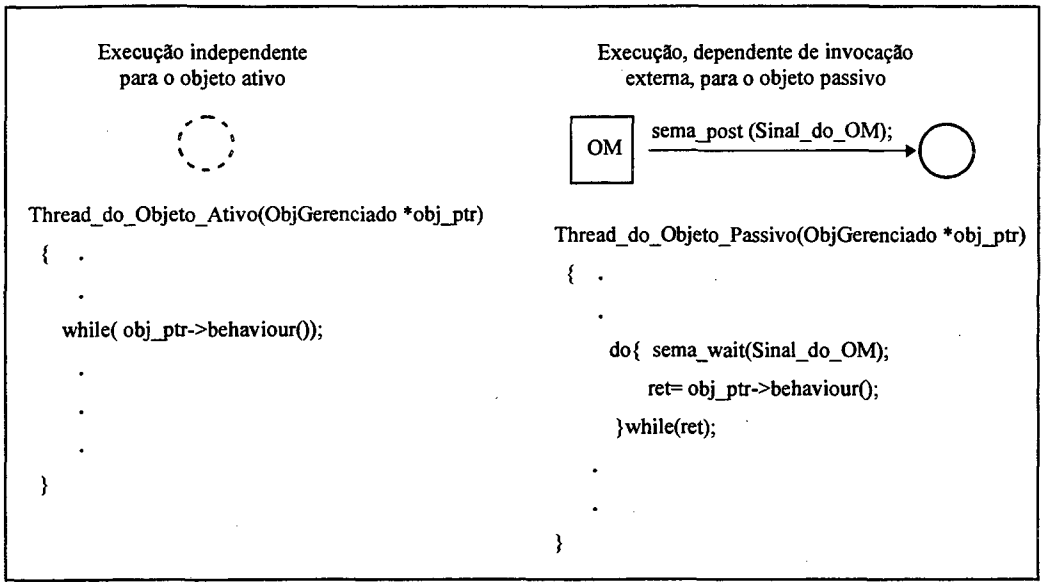

Figura 6.4 - Comportamento das threads em objetos passivos e ativos.

Como pode ser observado, objetos passivos executam seu comportamento mediante invocação do OM, a qual está sendo representada pelo incremento (*sema_post()*) de uma variável semáforo (Sinal_do_OM), utilizada pelo objeto gerenciado passivo. A *thread* para o objeto ativo executa constantemente seu comportamento.

Para o usuário desenvolvedor do comportamento do objeto passivo, ele somente deve saber que será invocado para o tratamento de uma requisição ou evento que necessite ser processado pelo seu comportamento, não precisando se preocupar com o código de sua ativação (Ex. sema_post()).

6.8 - Implementação do Núcleo *Multithreaded* Proposto

Para a implementação do modelo apresentado na seção anterior, foi utilizado o seguinte ambiente de desenvolvimento:

- PC 486 DX4-100
- 8 Mega bytes de RAM
- HD 850 Mega bytes
- Sistema Operacional LINUX 1.2.8
- Compilador C++ GNU 2.7.2

Em função do LINUX não implementar *threads* no *kernel* (até a versão 1.2.8 utilizada neste trabalho), o pacote utilizado trabalha no espaço do usuário.

O porte da implementação do núcleo para um ambiente diferente, que possua *pthreads* implementado (Ex. Solaris 2.5), é realizado sem nenhuma alteração no código, visto que a interface utilizada para a manipulação de *threads* segue o padrão POSIX [52].

Todo núcleo foi desenvolvido em C++, sendo que suas principais classes são:

- Classe OM - Implementa o comportamento do OM definido na seção 6.5;
- Classe MO - Representa os objetos gerenciados (objetos ativos/passivos);
- Classe MIT- Uma implementação da MIT. Oferece métodos para seu gerenciamento;
- Classe Queue - Implementa as filas de operações, respostas e notificações;
- Classe CoordS - Implementa o coordenador centralizado de sincronização.

Na implementação da classe Queue, foi necessário a utilização de mecanismos de exclusão mútua, a fim de controlar o acesso às filas de operações. O problema ocorre quando OM está inserindo operações na fila dos MOs, e estes manipulando a mesma fila (Extraindo operações da fila). Com a utilização destes mecanismos, é garantido o acesso exclusivo ao recurso compartilhado (a fila de operações) por uma das duas entidades ativas (OM/MO).

Este controle está internamente implementado na classe Queue, não sendo visível pelos objetos OM/MO. Portanto, usuários implementando parte dos MOs não necessitam se preocuparem com requisitos de exclusão mútua, para acesso aos recursos compartilhados, sendo este código oferecido pelo núcleo.

A Figura 6.5 apresenta um fragmento da implementação que utiliza exclusão mútua.

```
int QUEUE::insert(CMIP_PDU *ptr)
{
    if (pthread_mutex_lock(&Q_Sema))
    {
            printf("pthread_mutex_lock() ERROR (%s)\n",__FILE__);
            error_handler();
    }
    if ( n_elem == 0 )
    {
        Q=new OPER_QUEUE;
        if ( Q == NULL )
        {       printf("Erro na alocacao de memoria\n (%s) (%d)",__FILE__,__LINE__);
                error_handler();
        }
        Q_Begin=Q;
        Q->next=Q->prev=NULL;
    }
    else        // n_elem > 0
    {
        Q->next=new OPER_QUEUE;
        if ( Q->next == NULL )
        {       printf("Erro na alocacao de memoria\n (%s) (%d)",__FILE__,__LINE__);
                error_handler();
        }
        OPER_QUEUE *tmp;
        tmp=Q;
        Q=Q->next;
        Q->prev=tmp;
    }
    Q->elem=ptr;
    n_elem++;
    if (pthread_mutex_unlock(&Q_Sema))
    {
        printf("pthread_mutex_unlock() ERROR (%s)\n",__FILE__);
        error_handler();
    }
}
```

Figura 6.5 - Utilizando exclusão mútua para garantir à manipulação das filas (ThreadSafe).

Dentre os métodos da classe OM, o método principal é aquele que implementa seu comportamento (*OM::OM_Behaviour()*). Vale ressaltar, que na inicialização do agente, as três *threads* principais (OM, notificação, resposta) são criadas e inicializadas, e só terminarão sua execução caso o processo agente deixe de existir. No caso das *threads* associadas aos objetos, estas são criadas e destruídas dinamicamente, variando assim o nível de concorrência do processo agente. As Figuras 6.6 e 6.7 apresentam partes do código que implementam o comportamento do OM.

```
void * OM::OM_Behaviour(void *arg)
{
        MITnode *MO_ptr;
        CMIP_PDU *pdu;
        for(;;) // Para Sempre Faca:
        {
                if ( OM_Q.number_elements() > 0 )
                {
                      OM_Q.remove(&pdu);
                       switch  ( pdu->type ) {
                       case CREATE:
                              OM_op_create(pdu);
                              break;
                       case DELETE:
                              OM_op_delete(pdu);
                              break;

                       case ACTION:
                              OM_op_action(pdu);
                              break;

                       case GET:
                            SET:
                              OM_op_attr (pdu);
                              break;
                       case CANCEL_GET:
                              OM_op_cancel_get(pdu);
                              break;
                       default:
                              OM_op_error(pdu);

                      }
                }
                 else{  // Nao tem nada na fila
                           printf("Thr OM: Repassando\n"); // Debug
                           pthread_yield();
                }
        }
}
```

Figura 6.6 - Implementação do OM::OM_Behaviour().

```
OM::OM_op_create(CMIP_PDU *pdu)
{
        ...
        pthread_t  MO_thread_ID;
        //*** Cria FILA do Objeto
        QUEUE *MO_QUEUE_PTR=new QUEUE;
        if ( MO_QUEUE_PTR == NULL )
                {
                        // Código de tratamento de erro .......
                }
           //*** Cria Objeto
        MO  *MO_PTR=new MO(pdu->OID,MO_QUEUE_PTR);
         if ( MO_PTR == NULL )
                {
                // Código de tratamento de erro .......
                }
        //*** Cria Thread Para o Objeto
                if ( pthread_create(&MO_thread_ID,NULL,MO_Thread,MO_PTR) )
           {
                        // Código de tratamento de erro .......
           }
         // *** Cria entrada na MIT ***/
         if ( OM_MIT.insert_node(MO_PTR,MO_QUEUE_PTR) == -1 )
                {
                        // Código de tratamento de erro .......
                }
        send_notif(CREATE_OBJECT, MO_PTR);
}
```

Figura 6.7 - Seqüência de código executado para a operação CREATE.

Na definição do núcleo, foi especificado que os objetos ativos somente estarão consumindo recursos de execução quando realmente tiverem atividades a executar. Caso contrário, sua fatia de tempo (*timeslice*) não será utilizada por completo, sendo repassada a vez de execução para outras atividades do sistema. Isto é realizado através da operação pthread_yield().

Este comportamento pode incrementar consideravelmente a performance do sistema, principalmente em situações onde determinados objetos necessitam de um maior tempo de utilização da CPU. Um exemplo poderia ser objetos representando recursos com requisitos rígidos de monitoração e controle. A detecção de um evento e posterior emissão de uma notificação pode ser feita mais rapidamente, pois a CPU estará com um grau de disponibilidade maior para estas atividades. O fragmento de código da Figura 6.8, apresenta este comportamento, o qual efetua o repasse de execução caso não exista nenhuma operação a ser atendida.

```
// Thread que invoca o Comportamento do MO.
// Evita que o OM realize a interação diretamente com o método do Objeto,
// prevenindo o bloqueio do OM em função de faltas no MO.
// Esta função é um dos parâmetros do pthread_create(...).

void * MO_Thread(void * mo_ptr)
{
   MO *ptr=(MO *)mo_ptr;
   ptr->MO_Behaviour(NULL);
}

// Implementa o comportamento do MO.Código oferecido pelo núcleo.
void* MO::MO_Behaviour(void *mo_param)
{
        CMIP_PDU *pdu;
        for(;;)// Para Sempre Faça:
        {
          if ( Q_ptr->number_elements() > 0 )
          {
                Q_ptr->remove(&pdu);
                Exec_Operation(pdu);
          }
          else
          {
                printf("Thr_MO: Repassando\n"); // MSG de Debug
                pthread_yield(); // Repassa a vez de execução
          }
        }
}
```

Figura 6.8 - O método MO::MO_Behaviour e sua thread de execução.

Para efeitos comparativos, além do núcleo *multithreaded*, foi implementada uma versão de núcleo que se utiliza de objetos passivos e oferece uma interface síncrona com os objetos gerenciados. Este modelo foi adotado, pois é semelhante ao utilizado pela plataforma OSIMIS (seção 4.4.2.2). A comparação e análise dos resultados de cada implementação, será o assunto da próxima seção.

6.9 - Núcleo *Multithreaded* vs. *Callbacks*

Para efeitos de comparação, seria interessante avaliar a proposta apresentada neste trabalho com outras propostas *multithreaded* existentes atualmente. Contudo, após uma exaustiva pesquisa nos principais fóruns de discussão, livros/periódicos, *papers*, produtos comerciais, servidores de busca, e vários outras fontes de informação na área de gerência de redes, não foi encontrado nenhum material que apresentasse modelos de núcleos *multithreaded* para agentes CMIP. Em função disto, uma comparação com o modelo de *callbacks* foi realizada.

A primeira comparação é com relação ao conjunto de requisitos básicos que uma aplicação agente deve suportar. Nesta análise, serão utilizadas as plataformas estudadas na seção 4.4.2, as quais representarão o modelo de *callbacks*.

A Tabela 6.2, apresenta os requisitos suportados por ambos os modelos.

	Formas de Interações Suportadas								Interações Específicas com Recursos Reais											
	Ag-MO				MO-RR				Acesso p/ Demanda				*Cache* (*polling* periódico)				Alarmes do Recurso			
	S		A		S		A		S		A		S		A		S		A	
	B	N	B	N	B	N	B	N	B	N	B	N	B	N	B	N	B	N	B	N
Solstice ATK	■			■	■			■	■			■	■	■	■	■	■	■	■	■
OSIMIS	■				■			■	■				■			■	■			■
Núcleo *Multithreaded*		■		■		■		■		■		■		■		■		■		■
Desejável		■		■		■		■		■		■		■		■		■		■

Legenda:

■	Modo de operação da interface: **S**- Síncrono **A**- Assíncrono
■	Modo de Operação do Núcleo: **B**- Bloqueante **N**- Ñ Bloqueante
■	Serviços suportados pelo núcleo
	Serviços não suportados
■	Estes serviços não são suportados pelo núcleo básico. Contudo, o núcleo pode ser alterado facilmente para conseguir tal funcionalidade, o que na prática acontece. Esta alteração deve ser realizada pelo usuário.

Tabela 6.2 - Serviços e modos de operação suportados por ambos modelos.

A fim de se ter um referencial, foi inserido um modelo de núcleo desejável. O núcleo desejável suporta todas as formas de interações, além de oferecer ambas as interfaces de operação (Síncronas/Assíncronas). Neste modelo, tanto a interface síncrona como assíncrona não causam o bloqueio temporário na execução do processo agente. Todas estas características são consideradas desejáveis para o implementador de uma a aplicação agente.

O suporte de interfaces síncronas que não resultam no bloqueio do processo agente, é bastante útil, visto que o desenvolvedor do agente não necessita se envolver

com a complexidade de se trabalhar com chamadas de procedimentos assíncronos/Não-bloqueantes [35]. O desenvolvedor utilizando-se do modelo tradicional de programação síncrona, além de ser mais natural e de fácil operação, permite uma maior rapidez no processo de desenvolvimento (manutenção, depuração, etc.) do agente. Em função do núcleo *multithreaded* adotar o conceito de objetos ativos, este oferece a possibilidade de se ter interfaces síncronas, que não causam condições de espera temporária na execução processo agente.

O núcleo *multithreaded* também suporta qualquer um dos tipos de interações requeridas, e nos casos específicos de interações com RR, este deixa o usuário a vontade para implementar a interface com o recurso real. Qualquer modelo adotado pelo usuário, para a interação com o recurso gerenciado, não causará nenhuma mudança no comportamento das atividades do núcleo *multithreaded.*

A Tabela 6.2 deixa claro a maior flexibilidade oferecida pelo modelo de núcleo *multithreaded,* em comparação com o modelo orientado a *callbacks,* representado pelo OSIMIS e o Solstice TMN ATK.

Após verificar as funcionalidades suportadas em ambos os modelos, a seguir serão apresentados resultados de suas execuções. Toda coleta de informações dos agentes foi realizada internamente ao núcleo, descartando qualquer influência de protocolos (ROSE, ACSE, LPP, etc.) e camadas de comunicação. O objetivo é se avaliar os resultados do processamento interno do núcleo, visto que os outros componentes (protocolo de gerenciamento e camadas de comunicação) são padronizados, e se comportam da mesma forma para ambos os modelos.

A configuração dos dois agentes foi realizada adotando os seguintes parâmetros:

- Número de objetos: 30
- Número de requisições de operação: 60
- Tempo médio de interação com o recurso gerenciado: 2 segundos[1]
- Número de processos gerente: 1 gerente.
- Todas operações são confirmadas

[1] Este tempo foi adotado, considerando uma interação com um recurso fracamente acoplado ao sistema.

A implementação do núcleo orientado a *callbacks* oferece as mesmas funcionalidades que o modelo implementado pelo OSIMIS. Suas principais características são:

- Interface Agente-MO Síncrona
- Objetos Gerenciados implementados como Objetos Passivos
- *Single-Threaded*
- Interface Síncrona que exige o bloqueio do processo

Cada processo foi executado sobre o ambiente descrito na seção 6.8. Foram submetidas, aleatoriamente, operações de gerenciamento para cada um dos 30 objetos. Foi adotado como método de acesso ao recurso gerenciado, o acesso sobre demanda (*access-upon-external-request*), visto ser este o método mais utilizado em plataformas que seguem o modelo de *callbacks*.

O gráfico da Figura 6.9 apresenta os tempos de processamento para cada uma das requisições. O eixo X, contém o número da requisição. Foram realizadas 30 requisições, uma para cada objeto. No eixo Y, tem-se o tempo de resposta para as requisições, o qual está expresso em μs (micro segundos).

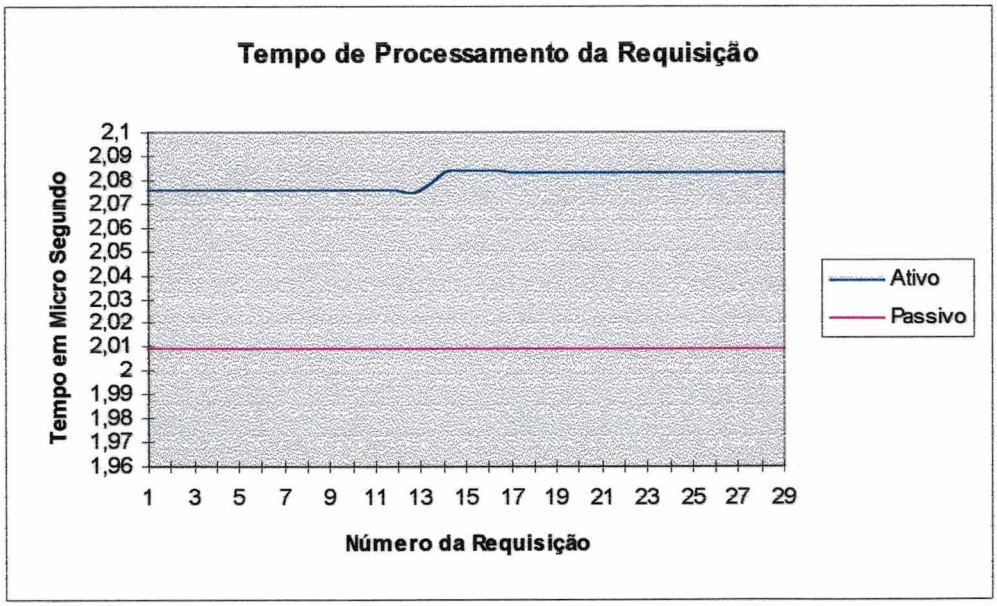

Figura 6.9 - Tempos de processamento para 30 requisições de gerenciamento.

Como pode ser observado, o tempo de resposta por requisição, para a implementação de objetos ativos (núcleo *multithreaded*) é um pouco maior que a

implementação de objetos passivos (*callbacks*). A diferença entre o maior tempo de resposta do núcleo com objeto ativos com o maior tempo utilizado pelo núcleo contendo objetos passivos é 0,074214u. Esta diferença existe, pelo fato do modelo de objetos ativos possuir o processamento extra para o gerenciamento das *threads*. Contudo, a latência de 0,074214u causada pelo processamento adicional é insignificante, se comparado aos benefícios oferecidos (ver Tabela 6.2).

Um detalhe interessante, é o aumento no tempo de processamento das requisições posteriores a décima terceira (13ª) requisição. Neste instante, provavelmente iniciou-se a chegada das várias notificações dos objetos, que estavam programadas para serem emitidas a qualquer momento (aleatoriamente) durante a execução do agente.

Como citado no Capítulo 4, modelos que se utilizam do paradigma de *callbacks*, e trabalham sobre um regime de acesso por demanda, por natureza não suportam a detecção de eventos nos recursos gerenciados, visto que sua interação com o recurso se dá no momento que um evento é gerado pelo gerente (chegada de uma requisição). Em função desta característica, foi gerado somente o gráfico de atendimento de notificações para o modelo de objetos ativos.

A Figura 6.10, apresenta o gráfico que representa o tempo que o agente leva para atender cada uma das 30 notificações geradas. Todas as 30 notificações foram geradas ao mesmo tempo. A geração de todas as notificações no mesmo instante foi utilizada, pois notificações geradas isoladamente eram rapidamente atendidas (aproximadamente 0,062056u). Com isso, resolveu-se exercer sobre o sistema agente, uma carga maior de notificações para serem atendidas, a fim de verificar seu comportamento em tal situação. A origem destas notificações foi realizada aleatoriamente.

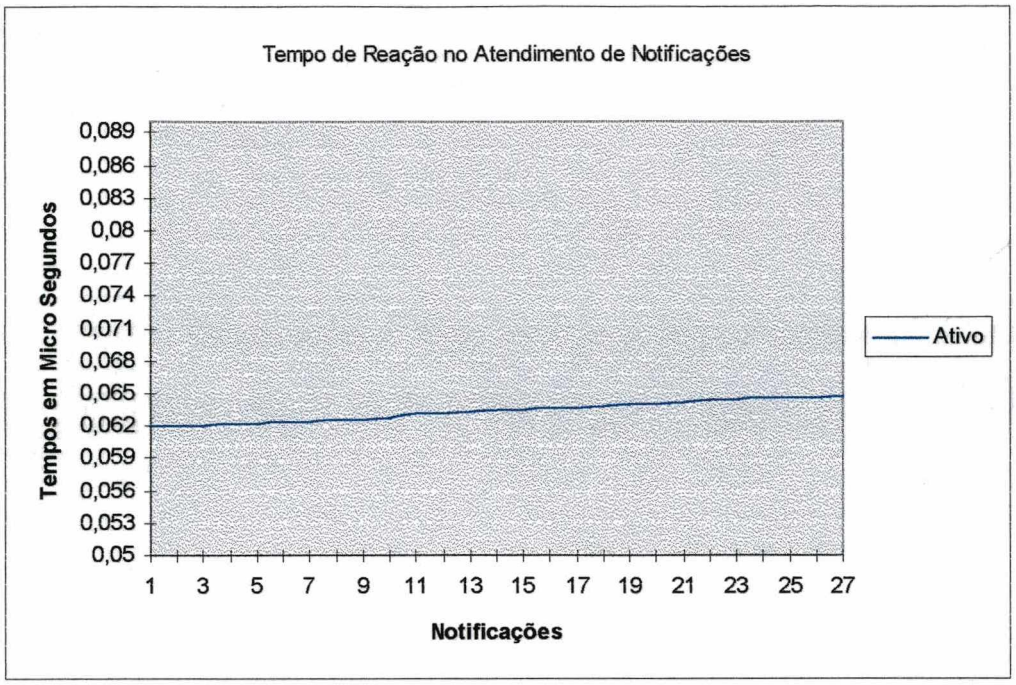

Figura 6.10 - Tempo utilizado para o atendimento de notificações.

Como ilustrado na Figura 6.1, notificações geradas pelo conjunto de MOs são enviadas para a fila de notificação, e são atendidas seguindo uma política FCFS (*first-come-first-served*) pela *thread* dedicada ao atendimento desta fila. O gráfico mostra, que para a primeira notificação gerada (que entrou na fila) o tempo de reação do agente foi aproximadamente 0,062056μ. Para a última notificação processada, desde o momento da sua criação até seu atendimento, levou-se aproximadamente 0,064755μ. O tempo total de atendimento das 30 notificações foi extremamente baixo (0,002699μ) Durante o atendimento das notificações, vale ressaltar que o agente estava concorrentemente executando outras atividades como processamento dos comportamentos dos MOs, atendimento de requisições recebidas do gerente, processamento da MIT, etc.

Finalmente, foi utilizado o utilitário `time`(1) do sistema UNIX, para efetuar a coleta do tempo de processamento total dos agentes. Este tempo é relativo ao recebimento das 30 requisições, processamento destas, juntamente com o processamento e emissão das 30 notificações (somente no modelo de objetos ativos), e envio dos resultados.

O utilitário `time` permite que se contabilize os tempos de utilização dos recursos de execução do sistema, por um determinado processo. Dependendo das

versões do comando `time`, seus resultados podem ser formatados e apresentados para o usuário de formas diferentes. A versão utilizada foi a GNU `time` 1.6 do Linux 1.2.8.

Os resultados do comando `time` para os dois agentes, são apresentados nas Figuras 6.11 e 6.12.

```
zeus% time agente_unthreaded
0.00user 0.03system 1:00.33elapsed 10%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (22major+59minor)pagefaults 0swaps
zeus%_
```

Figura 6.11 - Execução do agente com objetos passivos.

```
zeus% time agente_multithreaded
0.32user 1.91system 0:02.23elapsed 99%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (20major+149minor)pagefaults 0swaps
zeus%_
```

Figura 6.12 - Execução do agente com objetos ativos.

O tempo total de execução do processo agente, destaca-se sublinhado nas duas figuras. Para o modelo de objetos passivos, o tempo total de processamento foi aproximadamente 1 (um) minuto. Considerando 30 objetos para a MIB do agente, e tendo para cada objeto um tempo de execução de seu comportamento (MO-RR) de aproximadamente 2 segundos, obteve-se um tempo total aproximado de 1 (um) minuto (30x2s=60segundos).

Para o modelo *multithreaded*, o resultado do comando `time` demonstra que a execução concorrente das atividades do agente, possibilitam um aumento considerável na sua performance, pois durante a interação entre um MO-RR, outras atividades podem ser executadas ao mesmo tempo. Como resultado das execuções concorrentes se sobrepondo no tempo, tem-se um menor tempo de processamento do conjunto, resultando em apenas 2.23 segundos de execução.A Figura 6.13 ilustra a diferença.

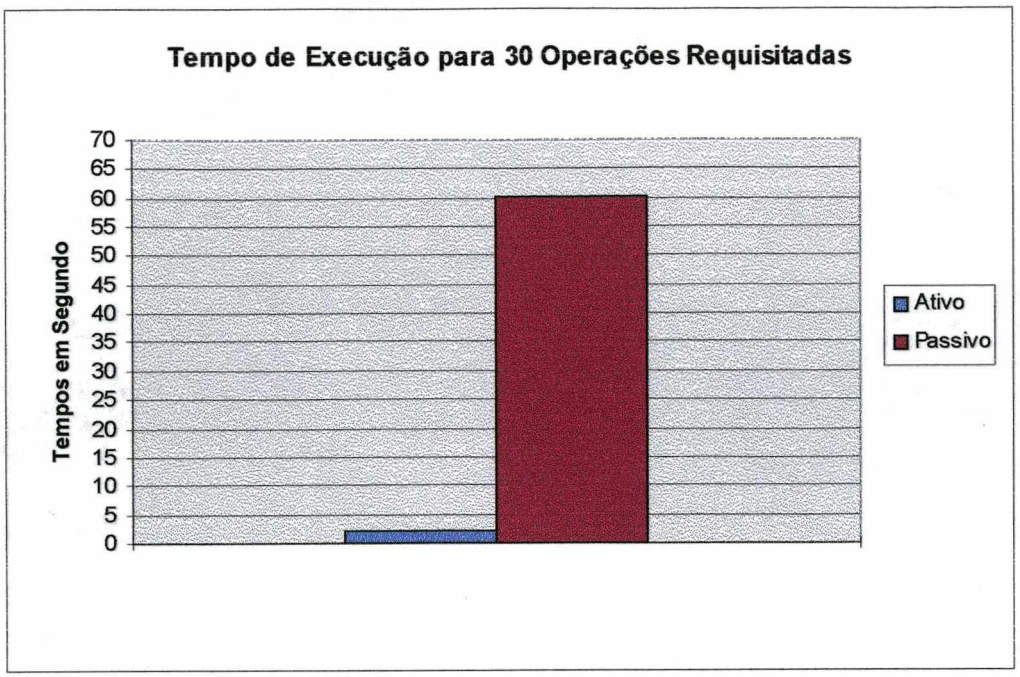

Figura 6.13 - Tempo total de execução para os dois modelos.

Esta grande diferença (58,1 segundos) entre os tempos de execução de ambos os modelos pode ser facilmente explicada através do diagrama apresentado na Figura 6.14.

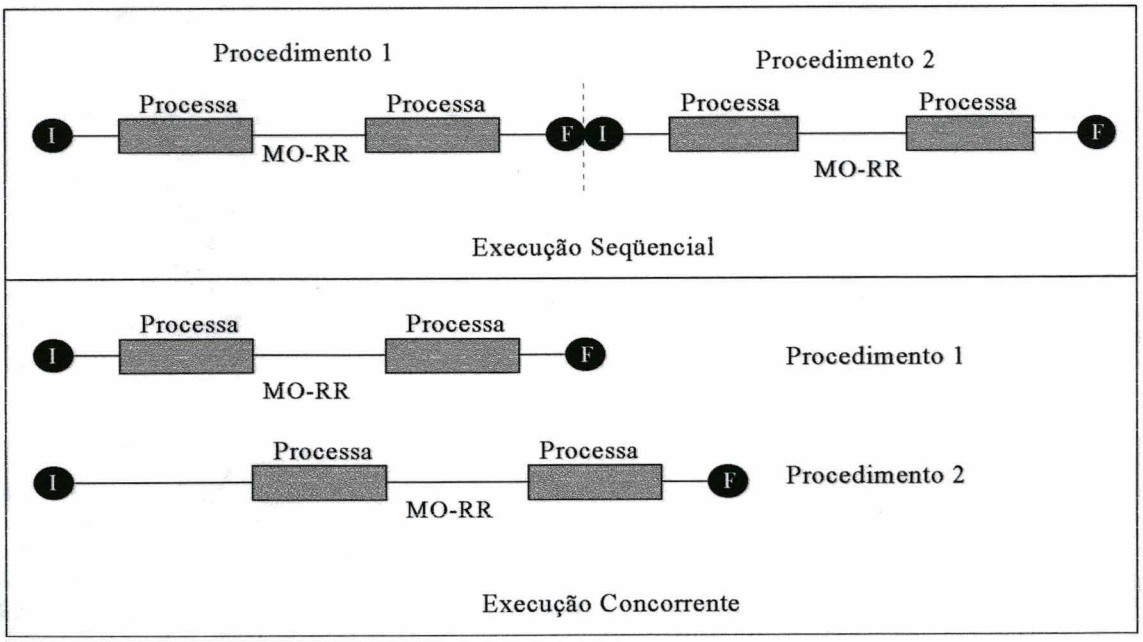

Figura 6.14 - Execução de atividades concorrentes e seqüenciais no tempo.

Como ilustrado, na execução seqüencial (modelo adotado para o agente *single-threaded*) existe somente um fluxo de execução no processo. Neste modelo, as execuções dos comportamentos dos objetos se realizam seqüencialmente. A execução

do comportamento, primeiramente realiza um determinado processamento (Ex. criação de estruturas de comunicação com o recurso real), comunica-se com o recurso gerenciado, e posteriormente executa mais algum processamento adicional (Ex. conversão das informações recebidas do recurso gerenciado para o formato definido no modelo de informação). Após o término da execução do objeto, outro objeto poderá ser ativado.

Já no modelo de execução concorrente (objetos ativos), a sobreposição da execução de outra atividade, no momento em que um objeto está interagindo com o recurso, é possível. Isto proporciona uma redução considerável no tempo de processamento final.

Para os casos onde não existem tempos de espera entre a interação MO-RR, e nenhuma parte do comportamento do objeto exige operações que causem o bloquei da execução (*delay*), o tempo de execução do modelo de objetos ativo é aproximadamente 8 segundos maior que o modelo de objetos passivo. As Figuras 6.15 e 6.16 apresentam os resultados de execução para este cenário.

```
zeus% time agente_unthreaded
0.00user 0.03system 1:00.33elapsed 90%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (22major+59minor)pagefaults 0swaps
zeus%
zeus%
zeus% time agente_threaded
 27.14user 0.21system 1:08.38elapsed 99%CPU (0avgtext+0avgdata 0maxresident)k
 0inputs+0outputs (21major+146minor)pagefaults 0swaps
 zeus%_
```

Figura 6.15 - Comando `time` para a execução com comportamento dos MOs sem pontos de espera

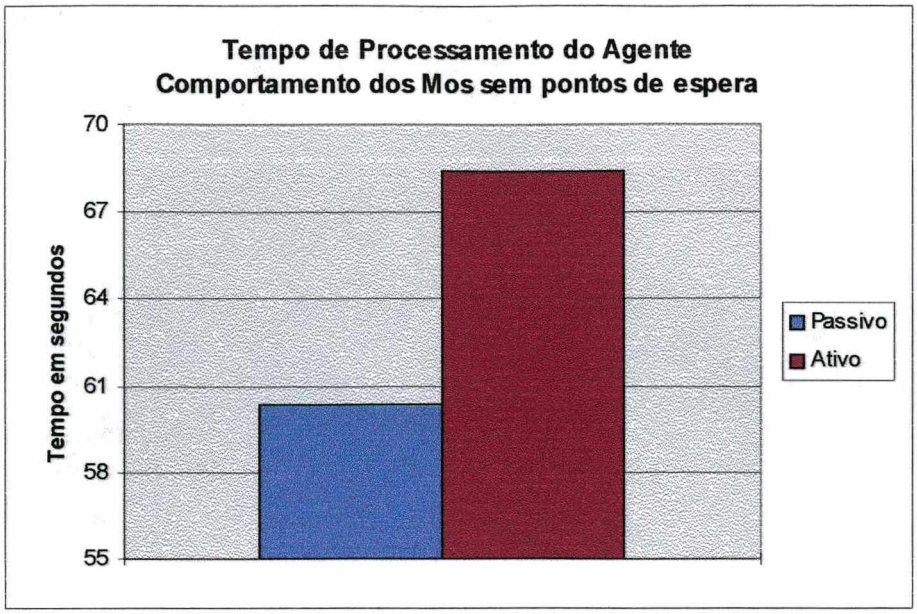

Figura 6.16 - Execução do comportamento dos MOs sem pontos de espera.

Foi realizada uma alteração no comportamento dos 30 objetos para que eles executassem uma computação sem nenhum requisito de I/O, ou qualquer outro processamento que causasse um bloqueio temporário na execução do agente (Ex. chamada de uma sub-rotina). Para efeitos de avaliação de ambos os modelos, foi inserido no comportamento dos objetos o seguinte fragmento de código (Figura 6.17).

```
....
for(int i=0;i<100;i++)
     for(j=0;j<60000;j++);
....
```

Figura 6.17 - Código inserido nos comportamentos dos objetos para coleta de tempos de execução.

Este código executa 100% de seu tempo continuamente, sem nenhum intervalo de espera. Sua execução, no ambiente em que foi realizado o teste, leva aproximadamente 2.30 segundos.

A execução sem pontos de espera, causou um aumento no tempo de execução do modelo de objetos ativos. Este aumento, já constatado no experimento da Figura 6.9, deve-se ao fato que agora a *thread* de cada objeto ativo está 100% de seu tempo em execução, o que não acontecia no modelo anterior (Figura 6.13).

Vale ressaltar, que o ultimo cenário apresentado, somente foi utilizado para fins de exploração de ambas implementações, visto ser de difícil ocorrência em situações reais. A modelagem e implementação de objetos gerenciados, por natureza,

exige que exista alguma forma de interação entre os MOs e os recursos que estão sendo representados. Tal interação, por mais simples que seja (Ex. chamada de sistema), provoca um tempo de espera no comportamento do objeto. Mesmo para aqueles objetos que somente existem para suporte ao gerenciamento (Ex. Log, Discriminador, etc.), o processamento de seu comportamento exige alguma forma de interação, causando algum tempo de espera em suas execuções (Ex. objeto log efetuando uma gravação no arquivo em disco).

Conclui-se com estes experimentos, que o modelo de objetos ativos oferece uma maior flexibilidade para a implementação dos agentes. Outro fator de grande importância, é o desempenho na execução das atividades intra-agente. Quando se tem objetos gerenciados representando recursos que exijam alguma forma de interação, o modelo de objetos ativos é extremamente eficiente, sendo bastante superior ao modelo de objetos passivos.

Quando se tem o comportamento dos objetos totalmente isento de situações de espera, situação bastante incomum, o modelo de objetos ativos é um pouco mais lento, no que se refere ao processamento das operações. Contudo, as funcionalidades oferecidas (Tabela 6.2) pelo núcleo *multithreaded*, justificam sua utilização em ambos os casos.

Estudos mais aprofundados de avaliação de desempenho, podem utilizar inúmeros cenários, possibilitando uma análise dos modelos em várias situações. As variações do número de objetos na MIB, do número de processos gerentes interagindo com o agente, das formas de interação (síncrona/assíncrona), das taxas de emissão de requisições, dentre outros fatores, são alguns exemplos do que se pode explorar na avaliação dos modelos.

Em [53], uma proposta para a avaliação de servidores de informação de gerenciamento é apresentada, a qual utiliza-se de modelagem e simulação para criar inúmeros cenários, submetendo-os a inúmeras situações. Neste é proposta uma análise das informações a respeito do comportamento dos modelos para cada um dos cenários avaliados.

CAPÍTULO VII

Conclusões

7.1 - Objetivos Alcançados

Dentro do projeto da plataforma de gerenciamento, um dos fatores que foi identificado como prioridade, foi a estruturação das aplicações agentes de gerenciamento. Após várias pesquisas, foi identificado que as várias plataformas de gerenciamento OSI, projetos comerciais e/ou acadêmicos, adotavam um modelo interno padrão, o qual oferecia uma arquitetura organizada para se agrupar os vários elementos e serviços que compõem as aplicações agentes. Esta arquitetura, denominada núcleo de agentes, foi então necessária, visto que sem ela não era possível integrar os vários trabalhos já realizados no âmbito do projeto da plataforma de gerenciamento OSI.

A necessidade de implementação do núcleo permitiu a pesquisa dos modelos atualmente utilizados, identificando seus possíveis problemas e limitações.

Após uma avaliação do estado da arte em desenvolvimento de agentes, tecnologias emergentes (*multithreaded*, objetos ativos, etc.) foram utilizadas como base para a construção do núcleo, permitindo que o modelo resultante eliminasse muitos dos problemas identificados nas implementações atuais.

Este trabalho, além de suprir a necessidade atual da plataforma, possibilitou a introdução de um novo conceito em pesquisas na área de construção de agentes; o desenvolvimento de agentes de alta performance.

Destaca-se também, a utilização do conceito de objetos ativos na implementação dos objetos gerenciados. Objetos ativos proporcionaram ao agente, menores tempos de resposta, juntamente com uma grande independência de execução de suas atividades internas. Uma das maiores vantagens desta independência, é a tolerância a faltas que possam vir a ocorrer em determinados objetos, o que não resulta na quebra de execução do processo como um todo.

Outra característica importante do modelo apresentado é que sua estrutura e implementação foram realizadas totalmente orientadas a objetos. A utilização de uma linguagem com suporte a OOP (no caso C++), também foi outro fator a se destacar, visto que as implementações atuais de núcleos, principalmente a nível comercial, são estruturadas mas não implementadas com linguagens orientadas a objetos [30] [54], dificultando a leitura e manutenção do código do agente.

7.2 - Dificuldades Encontradas

No decorrer deste trabalho, a maior dificuldade encontrada foi a falta de informações a respeito de modelos e implementações de núcleos de agentes de gerenciamento. Estas informações, normalmente fazem parte da implementação de produtos, os quais não são disponibilizados em domínio público.

A nível acadêmico, são poucos os projetos de plataformas de gerenciamento OSI. Nas poucas documentações encontradas, a parte que aborda o núcleo do agente na maioria das vezes é obscura, faltando informações importantes, que são vitais ao entendimento.

Este problema é causado basicamente pela falta de uma padronização a nível de núcleo de agentes, o que não se justifica, visto ser este um componente essencial para a criação de aplicações de gerenciamento no papel de agente.

Atualmente, o desenvolvimento de agentes CMIP é o centro das atenções em se tratando de telecomunicações em todo mundo. A padronização de um modelo para núcleos de agentes, permitiria uma maior eficiência na concepção destas aplicações, e tornaria a complexidade desta tarefa bastante reduzida.

7.3 - Futuros Trabalhos

A integração do núcleo apresentado neste trabalho, com pacotes que implementam as interfaces XOM/XMP, será prioridade para futuros trabalhos.

Atualmente, existem inúmeros produtos comerciais/acadêmicos que implementam XOM/XMP. Alguns destes oferecem além da interface, a implementação das camadas OSI, para fins de comunicação com plataformas de gerenciamento que seguem este modelo. Esta integração irá permitir a interoperação

mais fácil com outros sistemas de gerenciamento, além de oferecer um rico ambiente para testes, avaliações e aperfeiçoamento do núcleo *multithreaded.*

Complementando esta integração, são propostas para futuros projetos:

- A criação de um ATK (*Agent Tool Kit*) para a automatização do processo de geração do código do núcleo;
- A implementação de um mecanismo de *garbage collection* [55] para a remoção de objetos em falta na MIB do agente;
- A extensão do GDMO, para que na definição da classe de objetos gerenciados, seja possível definir se objetos daquela classe são ativos ou passivos;
- O porte do núcleo para outras plataformas, juntamente com estudos de desempenho sobre estas novas implementações;

As propostas anteriores, são alguns exemplos de trabalhos a serem realizados. Uma migração natural, será a implementação de aplicações gerentes que se utilizem da tecnologia de *multithreaded.* Nestas aplicações, a complexidade de implementação é bem menor que nos agentes, em função de suas características e requisitos. Contudo, os benefícios são grandes, visto sua natureza e tecnologias utilizadas atualmente (Ex. GUIs, SGBDs, etc.).

Outra grande perspectiva, é a utilização e avaliação deste modelo em máquinas multiprocessadas, onde pode-se explorar o paralelismo real de suas atividades.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] ISO/IEC 7498, Information Processing Systems - Open Systems Interconnection - Basic Reference Model, 1984.

[2] ISO/IEC 7498-4, Information Technology - Open Systems Interconnection - Management framework, 1992.

[3] ISO/IEC 10040, Information Technology - Open Systems Interconnection - Systems management overview, 1992.

[4] ISO/IEC 9596, Information Technology - Open Systems Interconnection - Common management information protocol - part 1: Specification, 1991.

[5] ISO/IEC 10165-4, Information Technology - Open Systems Interconnection - Structure of management information: Guidelines for the definition of managed objects, 1992.

[6] ISO/IEC 10165-1, Information Technology - Open Systems Interconnection - Structure of management information: Management information model, 1992.

[7] Stroustrup, B.; ELLIS, M. A. "C++: manual de referência comentado", Editora Campus, 1990.

[8] BRISA "Gerenciamento de Redes - Uma Abordagem de Sistemas Abertos", Makron Books, 1993.

[9] Uchôa, R. C.; Rodriguez, N. R. "Suporte para Monitoramento e Controle de Carga em Sistemas Distribuídos", SBRC'95, pp. 123-142, 1995.

[10] ISO/IEC 9595, Information Technology - Open Systems Interconnection - Common management information service definition, 1991.

[11] Tanenbaum, A. S. "Redes de Computadores", Editora Campus, 1994.

[12] BRISA "Arquiteturas de Redes de Computadores OSI e TCP/IP", Makron Books, 1994.

[13] ISO/IEC 10164-9, Information Technology - Open Systems Interconnection - Systems Management - Objects and Attributes for Access Control, 1991.

[14] ISO/IEC 10164-5, Information Technology - Open Systems Interconnection - Systems Management - Event report management function, 1992.

[15] Comer, D. E. "Internetworking with TCP-IP", vol. 1., Second Edition, Prentice-Hall, 1991.

[16] Sloman, M. "Policy Driven Management For Distributed Systems", Published in Journal of Network and Systems Management, Plenum Press, vol. 2 no. 4, 1994.

[17] Pavlou, G.; McCarthy, K.; Bhatti, S.; Knight. G. "The OSIMIS Platform: Making OSI Management Simple", University College London,1994.

[18] Silva, R.G.R.; Andrade, M.M.; Andrade, H.C.M.; Nogueira, J.M.S. "Metodologia de Desenvolvimento e Implementações de Agentes de Supervisão de Elementos de Rede para a Plataforma Distribuída SIS", SBRC'96, pp. 512-531, 1996.

[19] Raman, L. "CMISE Functions and Services", IEEE Communications Magazine, 1993.

[20] Yemini, Y. "The OSI Network Managment Model", IEEE Communications Magazine, 1993.

[21] May, K.; Ropellato, P; Matias, R.Jr.; Specialski, E.S. "Especificação e Implementação de Templates GDMO e Tipos Construtores ASN.1 para a Linguagem C++", II Congresso Argentino de Ciências da Computação (CACiC'96), San Luis, Argentina.

[22] Pozza, M. "Especificação e Implementação do Protocolo LPP", Projeto de Conclusão de Curso, INE/CTC/UFSC, 1995.

[23] Rocio, D.; Artifon, M.; Matias Jr., R.; Specialski, E.S. "Implementação da Função de Relatório de Alarme de Segurança para uma Plataforma de Gerência OSI", Aceito para apresentação no II Congresso Argentino de Ciências da Computação (CACiC'96), San Luis,Argentina.

[24] Oliveira, A.V. "Gateway CMIP-SNMP", Dissertação de Mestrado, CPGCC-INE/CTC/UFSC, 1995.

[25] De Mello, B.A. "Utilizando LOTOS na Concepção Formal de uma Aplicação para Gerência de Redes: Especificação e Verificação.", Dissertação de Mestrado, CPGCC-INE/CTC/UFSC, 1996.

[26] Borges, A.L.V.M. "Especificação e Implementação de Elementos de Serviços da Camada de Aplicação do Modelo OSI: ACSE e ROSE", INE/CTC/UFSC, 1995.

[27] Mansouri-Samani, M.; Sloman, M. "Monitoring Distributed Systems (A Survey)", Imperial College Research Report N°. DOC92/23, 1992.

[28] Pavlou, G. "Implementing OSI Management", 3rd International Symposium on Integrated Network Management, 1993.

[29] Bicalho, P.V.C.; Fayan, B.L. "Análise da utilização das API XOM/XMP no Desenvolvimento de Aplicações de Gerência para a Rede de Telecomunicações e Computadores", SBRC'96, pp. 495-511,1996.

[30] SunConnect - Sun Microsystems, Inc. "Solstice™ TMN Agent Toolkit 2.0 Programmer's Guide and Reference", Revision A, June/1997.

[31] ISO/IEC 10165-2, Information Technology - Open Systems Interconnection - Structure of management information: Definition of management information, 1992.

[32] Rose, M. T. "The Open Book: a practical perspective on OSI", Prentice-Hall, 1990.

[33] Comer, D.E.; Stevens, D.L. "Internetworking with TCP/IP - Volume III", Prentice-Hall, 1994.

[34] Powell, M.L.; Kleiman, S.R.; Barton, S.; Shah, D.; Stein, D.; Weeks, M. "SunOS Multi-thread Architecture", USENIX'91, 1991.

[35] Tanenbaum, A. S. "Distributed Operating Systems", Prentice-Hall, 1995.

[36] Stevens, W.R. "UNIX Network Programing", Prentice-Hall, 1990.

[37] Kleiman, S.; Shah D.; Smaalders, B. "Programming with Threads", SunSoft Press - A Prentice-Hall Title, 1996.

[38] Marchisio, L.; Ronco, E.; Saracco, R. "Modeling the User Interface", IEEE Communications Magazine, 1993.

[39] Bean, A.; Wood, D.; Fairclough, W. "Specifying Goal-Oriented Network Management Systems", IEEE Communications Magazine, 1993.

[40] Matias Jr., R.; Specialski, E. S. "A Multithreaded Core for Network Management Agents", ISCC'97 (IEEE Symposium on Computers and Communications), Julho/1997, Alexandria,Egito.

[41] Booch, G. "Object-Oriented Analysis and Design with Applications - Second Edition", The Benjamin/Cummings Publishing Company, 1994.

[42] Löhr, K. "Concurrency Annotations", OOPSLA'92, pp 327-340, 1992.

[43] Tsichritzis, D.; Nierstrasz, O.; Gibbs, S. "Beyond Objects: Objects ", IJICIS, vol. 1 no. 1, pp. 43-60, 1992.

[44] Bos, D.V.J.; Laffra, C. "PROCOL: A Parallel Object Language with Protocols", OOPSLA'89, 1989.

[45] Takashio, K.; Tokoro, M. "DROL: An Object-Oriented Programming Language for Distributed Real-Time Systems", OOPSLA'92, pp. 276-294, 1992.

[46] Matias Jr., R.; Specialski, E. S. "Managed Objects as Active Objects: A Multithreaded Approach.", IS&N'97 (4th International Conference on Intelligence in Services & Networks), Maio/1997, Como,Italia.

[47] ISO/IEC 10164-3, Information Technology - Open Systems Interconnection - Systems Management: Attributes for Representing Relationships, 1992.

[48] Ezhilchelvan, D. P.; Macêdo, A. R.; Shrivastava, S.K. "Newtop: A Fault-Tolerant Group Communication Protocol", Computing Laboratory, University of Newcastle, 1993.

[49] Shrivastava, S.K.; Dixon, G.N.; Parrington, G.D. "Objects and Actions in Reliable Distributed Systems", Computing Laboratory, University of Newcastle, 1987.

[50] Guerraoui, R.; Larrea, M.; Schiper, A. "Non Blocking Atomic Commitment with an Unreliable Failure Detector", Technical Report, Esprit Basic Research Project 6360, 1995.

[51] Babaoglu, Ö.; Fromentin, E.; Raynal, M. "A Unified Framework for Specification and Run-time Detection of Dynamic Properties in Distributed Computations", Department of Computer Science, University of Bologna, 1995.

[52] Gallmeister, B.O. "POSIX.4: Programming for the Real Word", O'Reilly & Associates, Inc., 1995.

[53] Sene Jr., I.G. "Comparação de Modelos de Agentes P/ Gerência de Redes Utilizando Simulação", Trabalho Individual, CPGCC-INE/CTC/UFSC, 1996.

[54] Serre, J.; Lewis, P.; Rosenfeld, K. "Implementing OSI-Based Interfaces for Network Management", IEEE Communications Magazine, 1993.

[55] Puaut, I. "A Distributed Garbage Collector for Active Objects", Proceedings of the 9th ACM-SIGPLAN, Portland, Oregon, 1994.