UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

UM MODELO PARA LINGUAGENS ORIENTADAS A OBJETOS DISTRIBUÍDO

por

GERSON GERALDO HOMRICH CAVALHEIRO

Dissertação submetida como requisito parcial
para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Philippe Olivier Alexandre Navaux
Orientador

Prof. Cláudio Fernando Resin Geyer
Co-orientador

Porto Alegre, Agosto de 1994.

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

CAVALHEIRO, GERSON GERALDO HOMRICH

UM MODELO PARA LINGUAGENS ORIENTADAS A OBJETOS DISTRIBUÍDO / GERSON GERALDO HOMRICH CAVALHEIRO.— Porto Alegre: CPGCC da UFRGS, 1994.
145 p.: il.

Dissertação (mestrado)—Universidade Federal do Rio Grande do Sul, Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, 1994. Orientador: Navaux, Philippe Olivier Alexandre; Co-orientador: Geyer, Cláudio Fernando Resin

Dissertação: Linguagens de Programação, Processamento Distribuído, Orientação a Objetos

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Hélgio Trindade
Pró-Reitor de Pesquisa e Graduação: Prof. Cláudio Sherer
Diretor do Instituto de Informática: Prof. Roberto Tom Price
Coordenador do CPGCC: Prof. José Palazzo Moreira de Oliveira
Bibliotecária-Chefe do Instituto de Informática: Zita Prates de Oliveira

Quer esteja eu louco ou frio,
obscecado por anjos
ou por máquinas
o último desejo é o amor.

A. Ginsberg

AGRADECIMENTOS

Chegando ao fim de mais uma etapa de trabalho, mais do que uma consquista pessoal, este documento representa a cooperação de várias pessoas que foram envolvidas no seu desenvolvimento.

Cooperação que veio dos colegas, como a Otilia, Soraia, Adenauer, Ceretta, De Rose, Krug, Ana, Roland, Menna, Zancanella, Paulo, Osório, Marinho, Leonardo entre outros, nos laboratórios, nas baias ou mesmo fora desta Universidade sempre prontos a discussões, técnicas ou não, procurando a melhor, ou muitas vezes a única solução para os infinitos problemas que surgiram neste período de tempo.

Meu reconhecimento também ao pessoal da manutenção dos laboratórios que estiveram sempre dispostos a colaborar, às secretárias do pós e do II resolvendo galhos e ao pessoal da biblioteca por muitas vezes se disporem a procurar *aquela* referência.

Aos professores Geyer e Caríssimi pelo espírito de cooperação.

Ao professor Navaux pela orientação e pela paciência e ao Rafael pela mão na massa e disposição.

Agradeço aos meus pais pela oportunidade que me deram, junto ao seu carinho e dedicação. Também as pessoas que estiveram comigo no início, no meio ou no fim de meu trabalho, acreditando no meu esforço e dando seu apoio para sua conclusão.

Também agradeço as inúmeras *concessões* que recebi enquanto me envolvia mais nas etapas mais duras de trabalho.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Linguagens de programação orientadas a objetos possuem diversas características que facilitam sua utilização frente a outras linguagens imperativas. No projeto e desenvolvimento de software, o mecanismo de herança permite a construção de sistemas na forma incremental e evolutiva, possibilitando a reutilização de códigos já escritos. Também é possível atingir aplicações com bons níveis de segurança e confiabilidade, através do encapsulamento de dados e funções sob forma de objetos, que também representam a unidade básica de execução em uma linguagem orientada a objetos.

O mesmo recurso que possibilita níveis elevados de segurança permite que linguagens orientadas a objetos sejam inerentemente distribuídas. Objetos possuem tanto área de dados e código de execução independentes dos demais. Acessos aos dados internos de um objeto somente são possíveis através de mensagens explícitas entre objetos. Neste caso um objeto solicita uma ação específica a outro objeto, podendo ser enviados parâmetros e existir retorno de resultados.

Este trabalho apresenta um modelo para construção de uma linguagem orientada a objetos distribuída. O ambiente para suportar a execução ao modelo é compostos por vários nodos de processamento com memórias locais individuais e contando com uma rede de comunicação para troca de mensagens entre os nodos. O modelo é discutido em dois níveis distintos: a nível de **linguagem** e a nível **operacional**. A nível de linguagem são analisados os recursos de programação normalmente utilizados em linguagens orientadas a objetos quando implementados em ambientes distribuídos. O ambiente de suporte à execução necessário ao suporte do modelo da linguagem é analisado pelo nível operacional.

A apresentação do modelo a nível de linguagem discute as características de uma linguagem orientada a objetos distribuída frente às

implementações seqüenciais convencionais. É ressaltada a implementação de herança em um ambiente de execução distribuído, que, não podendo ser através de compartilhamento, é efetuada através de cópia de código. Também são apresentadas novas diretivas de compilação necessárias exclusivamente a ambientes distribuídos. Tais diretivas visam explorar níveis de concorrência de uma aplicação durante sua execução, diferenciando classes que definem objetos locais ou distribuídos e diferentes tipos de mensagens entre objetos.

As formas de extrair o melhor desempenho nas aplicações e o gerenciamento do ambiente de execução são os pontos abordados pelo nível operacional do modelo. Em operação neste nível, um elemento de gerência de execução permite o controle tanto dos objetos da aplicação quanto dos nodos de processamento disponíveis para execução. A tarefa de controle de objetos viabiliza a criação e remoção de objetos durante a execução da aplicação, bem como a identificação de localização destes. O controle dos nodos de processamento possibilita analisar continuamente a carga computacional dos nodos de processamento. Assim, cada objeto a ser criado pode ser alocado em um nodo onde a carga computacional esteja baixa, propiciando um melhor desempenho no momento de execução da aplicação distribuído a carga entre os nodos.

A união do modelo de execução distribuído proposto a uma linguagem orientada a objetos resulta em uma linguagem eficiente tanto na produção de software como no desempenho de aplicações. A eficiência na produção de sistemas é obtida através de dois itens, a utilização do paradigma de orientação a objetos e a transparência do nível operacional para o programador, que não necessita conhecer os mecanismos utilizados para ativação de objetos e envio de mensagens. A eficiência de execução é obtida através da utilização de múltiplos nodos processadores servindo como base à execução.

Neste trabalho é também apresentado um protótipo para uma linguagem suportando o modelo distribuído proposto. A linguagem, denominada

DPC++ (Processamento Distribuído em C++), é voltada para execução em redes de estações de trabalho, sobre o sistema operacional Unix, utilizando sockets como mecanismo de comunicação. O estilo de programação em DPC++ é baseado em C++.

PALAVRAS-CHAVE: Linguagens Distribuídas, Orientação a Objetos, Processamento Distribuído

TITLE: "A MODEL FOR DISTRIBUTED OBJECT–ORIENTED LANGUAGES"

ABSTRACT

The objects-oriented programming languages have many features who make simple their use in front of others imperatives languages. In the software project and development, the inheritance mechanism allows an increasing and evaluative way of codes that have been written. It also possible gain applications with goods levels of security and confiability with the encapsulation of both data and functions in the form of object, which represent the basic execution unit in an object-oriented language.

The same resource that provides high levels of security also permits that object-oriented languages may be inherently distributed. Objects have their own area of data, their execution codes are independent from the other. Accesses to the internal data of an object are possible only through a specific protocol among objets. When this occurs, an object requests a specific action to other object with or without parameters or results return.

This work presents a model for build a distributed object-oriented language, devoted to environments compounded by several processing nodes with local memory and linked by a communication network. The model is discussed in two different levels: *language level* and *operational level*. In the language level are analyzed the programming resources usually used in object-oriented languages when implemented in distributed environments. The executing environments support are analyzed in the operational level.

In the language level presentation are made a discussion about distributed object-oriented language features in front of conventional sequential implementations. It is emphasized inheritance in a distributed executing envi-

ronment, who is done by code copy, due to can not be by memory sharing. Also are presented news compilation directives necessaries to the distributed environment. Those directives aim to explorer concurrence levels in an application during its execution, differing class who defining local or distributed objects and the different messages types among objects.

The operational level boards the ways that mean to extract the best performance for the applications and the execution environment management. An execution manager element allows the control as the application objects as the available to execution processor nodes. The task of object control makes possible the objects creation and removal during the application executing as well their network identification.

The processor nodes control allows the continuous analyzes of the computational load in the nodes available to processing. In this way, every object to be created can be allocated in a node with low occupation rates, propitiating a better performance in the application executing.

The union of the proposed distributed execution model to an object-oriented language results in an efficient language as in the software production as in execution performance. The systems production efficiency is obtained from two items: the utilization of the object-oriented paradigm an the transparency of the operational level to the programmer, that no need know the used mechanisms to object activation and message exchange. The execution efficiency is gained by the utilization of multiples processor nodes supporting the application executing.

In this work is presented a prototype that implements the proposed model. The language, called DPC++, Distributed Processing in C++, is turned to execute in workstation network with Unix operational system, using sockets as communication mechanism. The style of DPC++ programming are based in C++.

KEYWORDS: Distributed Languages, Object Oriented, Distributed Processing

1 INTRODUÇÃO

Uma análise das linguagens de programação, desde os primórdios da computação, mostra que sua evolução acompanha a disponibilidade de recursos de hardware e das aplicações que refletem as necessidades dos usuários. No início, manipulava-se pouca quantidade de dados e os computadores caros. Atualmente, a quantidade de dados envolvida nos processamento é grande, e não raro, compartilhada entre diversas máquinas.

Com o crescimento do volume de dados e da complexidade das tarefas envolvidas nos processamentos, a evolução das linguagens ficou caracterizada na busca de formas eficientes de manipulação de dados nos novos recursos computacionais disponíveis.

Os primeiros sistemas de computação, desenvolvidos pelos anos de 1938 a 1953 [HWA84], não apresentavam ao usuário propriamente uma linguagem de programação: o velho ENIAC era "programado" através de chaves. Esta forma rudimentar de programação era suficiente, uma vez que as operações realizadas pelo computador eram simples.

Uma primeira evolução na ciência da programação foi o surgimento de computadores programáveis em linguagens *assembly* no período de 1952 a 1963. O surgimento de computadores com capacidade e potencialidades maiores inviabilizou a "programação por chaves", ou programação em código binário. Uma linguagem baseada em mnemônicos, denominada **linguagem assembly**, invocando as operações desejadas foi introduzida nestes sistemas. Cada mnemônico nestas linguagens corresponde a uma tarefa específica suportada pelo elemento de processamento central. Assim, um programa em assembly "informa" ao computador, a cada instrução, uma operação a ser realizada.

A linguagem assembly provia um ambiente de programação ainda bastante restrito, apresentando porém uma enorme vantagem frente a codificação em código binário, além de ser suficiente para as aplicações nas quais os computadores eram voltados: realização de cálculos matemáticos. Os recursos de hardware eram bastante escassos, principalmente no que se refere à capacidade de memória, restringindo a abrangência de aplicações. Os programas eram curtos, manipulando pouca informação, realizando basicamente operações matemáticas.

O surgimento de memórias de anéis magnéticos (*magnetic core memory*), com maior capacidade de armazenamento, e o início da popularização do computador, com o barateamento dos custos de hardware, viabilizou o surgimento de linguagens de alto nível, como FORTRAN (*FORmula TRANslation*) em 1956, COBOL (*COmmon Business Oriented Language*) em 1959 e ALGOL (*ALGOrithmic Language*) em 1960. Nestas linguagens, uma instrução normalmente é traduzida em várias operações básicas do computador. O crescente aumento nos recursos disponíveis no computador também impulsionou o advento das linguagens de alto nível: se fazia necessário uma maior abstração do hardware no momento da programação.

No que concerne ao desenvolvimento de computadores, o período de 1962 a 1975 foi marcado principalmente pelo:

- crescimento tecnológico utilizado para implementação do hardware, tornando os elementos de processamento mais rápidos e as memórias com maiores capacidades;
- barateamento dos custos dos equipamentos, com uso de tecnologias menos dispendiosas e da produção em série de máquinas;
- início da popularização do uso do computador;
- aumento do poder de processamento e dos recursos disponíveis, tornando mais complexa a tarefa de controle do hardware;

- desenvolvimento de computadores para uso em aplicações específicas.

Muitas linguagens de alto nível foram desenvolvidas neste período, como as de uso genérico Pascal e C, em 1970 e 1972 respectivamente, e as de uso voltado a uma área específica, como Lisp e Prolog, em 1963 e 1972, linguagens desenvolvidas para uso em sistemas inteligentes e de processamento de conhecimento.

Nos anos 70, ficou caracterizado o uso intensivo de técnicas de programação estruturada a qual visa o desenvolvimento independente de módulos, aumentando a eficiência de desenvolvimento e a confiabilidade de programas. As aplicações já não consistiam em apenas operações matemáticas, uma quantidade maior de dados era manipulada, e os programas se tornaram mais extensos. Existia a necessidade de um controle maior no desenvolvimento de software, diversas metodologias de projeto foram desenvolvidas, sendo Pascal a linguagem que melhor representava os conceitos de "programação estruturada", uma abordagem para produção sistematizada de software.

A partir de 1975, os sistemas computacionais passaram a sofrer alterações. Já não se fala apenas em arquiteturas Von Neumann, entram em cena arquiteturas não convencionais de computadores [HWA84], como as arquiteturas vetoriais, multiprocessadoras e redes de nodos de processamento. As linguagens acompanharam este desenvolvimento, algumas das já existentes absorveram características dos novos hardwares disponíveis, formando novos dialetos. Outras linguagens foram especialmente desenvolvidas para os novos ambientes.

Tal como verificado a nível de hardware, a complexidade de desenvolvimento de software também cresceu. Sendo levada a extremos, o estilo de programação e desenvolvimento estruturado culminou no paradigma de orientação a objetos [TAK90], que reúne facilidades para o desenvolvimento de

aplicações com alto grau de abstração de dados. Estes conceitos foram absorvidos por diversas linguagens, ditas orientadas a objetos, como SMALLTALK e C++.

No que se refere ao projeto de software, alguns autores, como [YAU92], apontam o paradigma de orientação a objetos como o mais promissor para desenvolvimento de sistemas distribuídos, devido a sua capacidade para expressar o paralelismo e potencial de manutenção e reutilização de software. Diversas propostas de linguagens orientadas a objetos concorrentes foram implementadas, algumas encontram-se sumarizadas em [WYA92].

Este trabalho apresenta um modelo para execução distribuída de objetos em uma rede de nodos independentes de processamento. Com a introdução deste modelo em uma linguagem orientada a objetos, espera-se obter ganhos de desempenho de aplicações pelo uso de ambientes de execução distribuídos utilizando um ambiente de programação que possua facilidades de desenvolvimento de sistemas.

No capítulo 2 é discutida a utilização de sistemas distribuídos. São abordados temas referentes à configuração de hardware e características de linguagens de programação distribuídas.

O capítulo 3 apresenta conceitos referentes ao paradigma de orientação a objetos segundo o ponto de vista de linguagens de programação. São também apresentadas e discutidas características de implementação de linguagens orientadas a objetos em ambientes concorrentes e distribuídos.

O modelo proposto neste trabalho é apresentado no capítulo 4, sendo analisado a nível operacional e a nível de linguagem. Este capítulo também apresenta o ambiente de programação a ser utilizado em conjunto com o modelo proposto.

O capítulo 5 apresenta os recursos utilizados para implementação de uma linguagem protótipo suportando o modelo em um ambiente de estações de trabalho. Esta linguagem, denominada DPC++ – *Processamento Distribuído em* C++, é uma extensão de C++ suportando a execução distribuída de objetos. No final deste capítulo é apresentada uma aplicação desenvolvida em DPC++ e apresentados índices de desempenho.

A conclusão deste trabalho é apresentado no capítulo 6, sendo comentadas as características do modelo proposto e da linguagem DPC++.

Ao final são apresentados 4 anexos. O primeiro discute algumas características gerais de C++. Os demais que ilustram a implementação do protótipo contendo parte do código referente ao ambiente de execução da aplicação proposta no capítulo 5.

2 NOVOS AMBIENTES DE PROGRAMAÇÃO

Sistemas distribuídos consistem basicamente de módulos de uma aplicação executando em elementos processadores independentes. Estes módulos compartilham informações através de mensagens para solução de um único problema. O surgimento dos sistemas distribuídos ocorreu simultaneamente à ploriferação de arquiteturas multicomputadoras e redes de computadores, caracterizando uma alternativa de menor custo aos supercomputadores. Os primeiros sistemas eram programados em linguagens tradicionais, utilizando bibliotecas de comunicação, fornecidas pelo sistema operacional, para realizar a cooperação entre processos.

Estes ambientes ainda são encontrados atualmente, provendo porém poucos recursos para programação. A tarefa do programador é bastante complexa, cabendo-lhe, além de codificar a aplicação, controlar a comunicação entre as partes do programa. Deve ser monitorado de forma explícita o envio e recebimento de mensagens. A montagem e desmontagem de pacotes de mensagens também é realizada de forma manual, possibilitando que a estruturas de dados internas a programas sejam enviadas pelo protocolo de comunicação. Neste ponto o programador deve interferir diretamente com o protocolo de comunicação, realizando a conversão de dados internos ao programa em dados manipuláveis pelo protocolo de comunicação para o envio de mensagens e o a tarefa inversa no recebimento.

Hoje a tarefa de programação de sistemas distribuídos está bastante simplificada com o uso das diversas linguagens distribuídas disponíveis. [BAL89] classifica estas linguagens em duas categorias: linguagens distribuídas com espaço de endereçamento logicamente compartilhado e linguagens distribuídas com espaço de endereçamento logicamente distribuído. Em ambas categorias, o

suporte de hardware pode ser fisicamente distribuído como não distribuído, cabendo o controle de interação entre os processos à implementação da linguagem.

Nesta mesma classificação, as linguagens encontram-se agrupadas em classes segundo o mecanismo de comunicação utilizado. A tabela 2.1 reproduz a classificação apresentada por [BAL89].

Em qualquer um dos casos, a programação com linguagens distribuídas difere da programação em linguagens seqüenciais por utilizar múltiplos processadores cooperando entre si, existindo a possibilidade da ocorrência de uma falha parcial de hardware, tornando inoperante (ou com operação incorreta) um ou um grupo de processadores.

Tabela 2.1: Uma classificação para linguagens distribuídas

Endereçamento Distribuído	**Endereçamento Compartilhado**
mensagens assíncronas mensagens síncronas *Rendezvous*	linguagens funcionais
RPC primitivas múltiplas	linguagens lógicas
objetos transações atômicas	estruturas de dados distribuídas

As próximas seções deste capítulo abordam a conceituação de sistemas distribuídos, os recursos necessários para sua implementação e os pontos considerados para o projeto de uma linguagem distribuída.

2.1 Sistemas Distribuídos

Visando obter um melhor desempenho na execução de operações, foram buscadas alternativas para o modelo de computador Von Neumann. Em ambientes monoprocessador foram introduzidos conceitos de multiprogramação e uso de técnicas *pipeline* [KIR91].

Em ambientes com diversos processadores, surgiram os computadores matriciais, máquinas de redução e orientadas a fluxo de dados, multiprocessadores e multicomputadores [BAL89]. Enquanto os computadores matriciais executam a mesma instrução sobre um conjunto diferente de dados, as máquinas de redução e orientadas a fluxo de dados executam diferentes operações sobre diferentes conjuntos de dados ao mesmo tempo.

Multiprocessadores e multicomputadores são bastante semelhantes: ambos executam diferentes tarefas sobre conjuntos diferentes de dados, porém, multiprocessadores compartilham uma área de memória comum, enquanto multicomputadores possuem memória distribuída entre os diversos processadores interligados por uma malha de comunicação.

Esta diferença é bastante significativa no que se refere à cooperação entre as tarefas, caso estas devam resolver um mesmo problema. No caso de multiprocessadores, a memória serve como elemento de ligação, comunicação e sincronismo entre os processadores. Nos multicomputadores, estas operações são realizadas por meio de troca de mensagens entre os processadores independentes através da malha de interconexão.

A existência de concorrência é resultado da utilização de paralelismo, simultaneidade ou *pipeline* de tarefas na execução de aplicações. O paralelismo consiste em diferentes eventos ocorrendo em múltiplos processadores em um mesmo intervalo de tempo, a simultaneidade permite que estes eventos ocorram em um mesmo intervalo de tempo e o *pipeline* permite a execução sobreposta de partes de diversos evento.

Estendendo este conceito, a utilização de sistemas distribuídos também consiste em uma forma de exploração de concorrência, pois os sistemas distribuídos baseiam-se na execução cooperativa e simultânea de processos. Esta afirmação é especialmente verdadeira considerando que, com o avanço das tecnologias de comunicação, as distâncias entre o processamento paralelo, provido por

multiprocessadores, e distribuído, multicomputadores e redes de computadores, estão diminuindo [HWA84].

A figura 2.1 identifica a abrangência do termo concorrência, desde de ambientes monoprocessadores a sistemas de processamento distribuído. Em cada um destes sistemas existe em comum múltiplas tarefas sendo executadas por processos com fluxos de execução próprios.

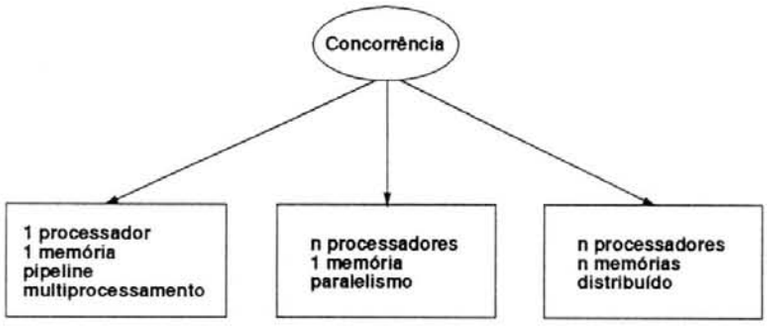

Figura 2.1: Tipos de concorrência

Para ambientar o problema de processamento distribuído, neste trabalho é utilizada a mesma definição de sistema distribuído adotada por [BAL89]:

> "Um sistema computacional distribuído consiste de múltiplos processadores autônomos que não compartilham memória primária, mas cooperantes através de uma rede de comunicação."

Segundo esta definição, cada processador em um sistema distribuído executa seu próprio fluxo de instruções sobre seus dados locais, devendo tanto os dados como o programa estar armazenados em sua memória privada. Eventuais trocas de informação entre os processadores são realizadas através de mensagens enviadas por uma rede de comunicação que os interliga.

Uma vasta gama de arquiteturas, como multicomputadores, redes locais de nodos de processamento (LANs - *Local Area Networks*) e redes de lon-

gas distâncias (WANs - *Wide Area Networks*), enquadram-se perfeitamente nesta definição. Estas arquiteturas são classificadas em forte ou fracamente acopladas, conforme a malha de comunicação.

Processadores interligados por uma rede confiável e rápida, com tempos de transmissão de mensagens na ordem de milisegundos, são considerados sistemas distribuídos **fortemente acoplados**. Este é o caso das arquiteturas multicomputadores, compostas, por exemplo, de um conjunto de Transputers, como a máquina T-Node [FLI89].

WANs compõem o outro extremo: sistemas distribuídos **fracamente acoplados**. A rede de comunicação nestes sistemas possui baixa confiabilidade e velocidade de transmissão bastante lentas, com tempo de transmissão de mensagens na ordem de segundos. Bons níveis de integridade, garantindo o recebimento correto de mensagens, pode ser obtido através de protocolos de comunicação adequados.

Em uma posição intermediária entre os sistemas forte e fracamente acoplados encontram-se as configurações em LANs, possuindo velocidades e confiabilidade melhores que nas WANs, mas ainda inferiores às encontradas em multicomputadores. Este é o caso das redes locais de estações de trabalho e/ou de computadores pessoais.

A opção por sistemas fraco ou fortemente acoplados depende em muito do grão de processamento da aplicação, ou seja, da quantidade de processamento realizado por um nodo entre cada envio de mensagem. **Granulosidade fina** de processamento implica na necessidade de muita comunicação entre os nodos, justificando a necessidade de uma malha de comunicação eficiente. Já uma rede não muito veloz pode vir a ser suficiente para aplicações com **granulosidade grossa**, cujo custo de comunicação é bastante alto.

2.1.1 Utilização de sistemas distribuídos

Segundo [BAL89], a utilização de sistemas distribuídos é motivada pela necessidade de implementar aplicações em que seja buscado pelo menos uma das características abaixo:

- melhora de desempenho;
- tolerância a falhas;
- especializações funcionais; ou,
- algoritmos tipicamente distribuídos.

Melhorar o desempenho de aplicações, diminuindo seu tempo de execução, é uma das razões mais fortes para uso de sistemas distribuídos e de sistemas paralelos. A redução do tempo de execução de uma aplicação é obtida através do seu particionamento em tarefas alocadas a diferentes processadores. Muitas aplicações na área de computação gráfica, como processamento de imagens, utilizam esta técnica.

Outras aplicações utilizam replicação de dados e/ou de processamento como forma de garantir a integridade dos dados manipulados e das operações realizadas. Aplicações típicas de tolerância a falhas são encontradas em sistemas para aviões e bancos de dados distribuídos.

Sistemas distribuídos podem também ser configurados com processadores dedicados a tarefas específicas, portando-se como prestadores de serviços. A interação nestes sistemas se dá pela requisição dos serviços prestados pelos diferentes processadores. O sistema operacional Amoeba [MUL90] possui esta característica, utilizando diversos servidores, como servidor de arquivo e servidor de impressão instalados em nodos de processamento específicos.

Aplicações como correio-eletrônico são exemplos de aplicativos inerentemente distribuídos. Uma carta-eletrônica é enviada de um nodo origem a um destino, passando por vários nodos de tratamento intermediário, sendo o processo de encaminhamento da carta ao destino especificado realizado de forma independente nos diferentes nodos.

2.2 Suporte ao Processamento Distribuído

Segundo [BAL89], os principais itens a serem analisados para o projeto e desenvolvimento de linguagens distribuídas são:

1. a forma de expressão do paralelismo a ser utilizado pela linguagem;
2. os mecanismos de comunicação e sincronização entre as partes da aplicação; e,
3. o esquema de suporte a falhas parciais no sistema.

O nível de detalhamento em cada um dos itens acima, bem como a abordagem adotada, depende do tipo de aplicação a qual a linguagem se destina. O restante desta seção discute alguns pontos relevantes à análise dos itens acima.

2.2.1 Expressão do paralelismo

A execução de partes da aplicação simultaneamente em diferentes nodos de processamento é requisito essencial a ser provido pelas linguagens distribuídas. A forma de expressão do paralelismo denota a unidade de concorrência, ou seja, o grão de processamento suportado pela linguagem. Portanto, a opção entre as diferentes formas de expressão de paralelismo é também fortemente

influenciada pelo hardware sobre o qual a linguagem deve ser implementada, paralelismo expresso em grandes blocos são voltados às redes de computadores, paralelismo expresso em grãos pequenos às máquinas multicomputadoras.

As formas de expressão de paralelismo atualmente suportadas pelas linguagens distribuídas podem ser classificadas em cinco grupos [BAL89]. Estes grupos encontram-se listados abaixo em ordem de granulosidade, de grossa à fina, com exemplos de linguagens que as implementam.

- Processos, p. ex. Concurrent C [ROO86].
- Objetos, p. ex. Distributed Smalltalk [BEN87].
- Comandos, p. ex. Occam [INM84].
- Expressões, p. ex. ParAlfl [HUN86].
- Cláusulas, p. ex. Concurrent PROLOG [SHA87].

Processos

Com esta forma de expressão de paralelismo, os processos são considerados como elementos de processamento autônomo. Cada processo possui sua própria área memória contendo dados e código. Dentro de um processo a apenas um fluxo de execução seqüencial. A declaração de processos é feita de forma semelhante à declaração de procedimentos em linguagens de programação tradicionais.

A criação de processos pode ser *implícita* ou *explícita* através do uso de um construtor do tipo *create processo*. Em algumas linguagens com criação implícita de processos, o número de processos a serem criados é definido em tempo de compilação, permitindo um melhor aproveitamento dos recursos dis-

poníveis, restringindo porém a flexibilidade da linguagem. Nas linguagens que suportam criação dinâmica, deve ser provido um mecanismo para promover o estabelecimento de canais de comunicação entre diferentes processos.

Normalmente um processo termina sua execução por si mesmo, mas as linguagens podem dispor de primitivas de remoção de processos.

Objetos

Um objeto encapsula dados e comportamento, podendo ser utilizado como unidade de paralelismo. Em linguagens seqüenciais é implementado o modelo de objeto passivo, ou seja, apenas um objeto está no estado ativo em uma determinada unidade de tempo. Neste modelo um objeto é ativado quando recebe uma mensagem; o objeto originador da mensagem permanece bloqueado, ou seja inativo, enquanto não receber resposta à sua mensagem; ao receber a resposta, o objeto originador da mensagem volta ao estado ativo e o recipiente ao estado passivo.

A extensão do modelo de objetos passivos para permitir a execução paralela de eventos em uma linguagem concorrente pode ser obtida através da inclusão de pelo menos uma das características abaixo:

- permitir que o objeto esteja ativo sem ter recebido uma mensagem;
- permitir que o objeto continue ativo após enviar resposta a uma mensagem recebida;
- permitir o envio de uma mensagem a um grupo de objetos (*broadcast* de mensagens); ou,

- permitir o envio de mensagens assíncronas entre objetos, possibilitando a execução paralela entre os objetos origem e destino da mensagem.

Comandos paralelos

Comandos paralelos consiste em uma estrutura bastante simples onde comandos de paralelização identificam grupos de instruções a serem executadas de forma paralela. Este método não é apropriado para construção de grandes sistemas. A forma geral desta estrutura é:

EXECUTE_PARALELO
Comando_1
Comando_2
FIM_PARALELO

Neste exemplo, Comando_1 e Comando_2 executam suas tarefas em paralelo, não havendo comunicação entre os processos. O sincronismo é reestabelecido ao final do bloco paralelo, onde a execução prossegue somente quando todas tarefas paralelas terminarem.

Expressões

O paralelismo de expressões é aplicado em linguagens funcionais, baseadas em expressões matemáticas. Nestas linguagens, o resultado da aplicação de uma função depende unicamente de seus dados de entrada, não havendo também qualquer efeito colateral, pois não há variáveis globais nem manipulação

direta de memória. Com estas características a introdução de concorrência nas funções pode ser facilmente implementada.

Por exemplo, na expressão $H(F(3,4),G(8))$ é irrelevante se a função F ou G for executada primeiro, ou mesmo se ambas executam de forma paralela. A única restrição é que uma função somente possa ser disparada quando todos os dados de entrada estiverem disponíveis. Neste exemplo, H somente será executada quando F e G terminarem suas computações, indicando o sincronismo entre as funções pelos parâmetros necessários.

A princípio, qualquer função pode ser executada de forma distribuída, tendo seu próprio processo, implicando em paralelismo implícito de granulosidade fina. Deve ser considerado o custo de distribuir todas as funções como processos independentes, uma vez que quando estas forem voltadas a execução de tarefas simples, como adição de dois inteiros, pode ser mais viável executá-la localmente. Os compiladores disponíveis ainda não implementam boas técnicas para obtenção da máxima eficiência neste tipo de paralelismo.

Cláusulas

As linguagens lógicas provêem pelo menos duas estruturas para suportar a execução paralela de tarefas:

1. paralelismo intra-cláusulas (paralelismo **E**); e o,
2. paralelismo inter-cláusulas (paralelismo **OU**).

Por exemplo, dadas duas cláusulas para A:
$A :- B, C, D.$
$A :- E, F.$

ambas podem executar em paralelo até uma das duas obter sucesso (resultar verdadeiro para A) ou ambas falharem (resultando que A é falso). Internamente, os subteoremas das cláusulas também podem ser computados em paralelo, até um falhar ou todos resultarem em sucesso.

Programas lógicos paralelos podem, a exemplo dos programas em linguagens funcionais paralelas, recair em processos dedicados a tarefas simples, cujo custo de distribuição torna-se maior que a execução local do processo. Outra característica da distribuição do processamento é a dependência entre cláusulas devido a conflitos por uso de variáveis compartilhadas. Há métodos automáticos para detecção de dependência e formas de contorná-las, como executar seqüencialmente as cláusulas com dependência mútua.

2.3 Cooperação entre Processos

Para permitir que os diferentes processos de uma mesma aplicação possam cooperar entre si é necessário que a linguagem disponha de mecanismos de comunicação e de sincronismo. Através destes mecanismos, um processo pode solicitar a outro alguma informação ou a realização de alguma tarefa. Conforme o caso, o processo solicitante deve aguardar o retorno de algum dado ou o término da operação solicitada.

As linguagens distribuídas apresentam duas formas de comunicação entre processos: baseada em *compartilhamento de dados*[1] e *envio de mensagens*. No primeiro caso, as implementações clássicas encontram-se implementadas como:

- Estruturas de dados compartilhadas. Uma área de dados, acessível por todos os processos, mantém elementos de dados (denomina-

[1] Em implementações distribuídas a linguagem provê a simulação do uso de memória compartilhada.

dos *tuplas*), a qual é acessada através de três operações atômicas: *read*, lê o conteúdo de uma posição da memória compartilhada; *in*, lê e retira o conteúdo de posição da memória compartilhada; e, *out*, insere uma informação na memória compartilhada. Uma das implementações desta forma de comunicação é encontrada em Linda [AHU86]; também utilizada como base para extensões de linguagens, como em C-Linda [NET92], permitindo a programação em redes de estações de trabalho.

- Compartilhamento de variáveis lógicas. Uma variável lógica é utilizada como um canal de comunicação entre processos dedicados a prova de subteoremas. Implementações são encontradas em linguagens baseadas em PROLOG, como Concurrent PROLOG [SHA87] e PARLOG [GRE86].

Outra forma de realizar a cooperação entre processos é através da troca de mensagens, envolvendo, de um lado, o processo origem da mensagem e de outro, o processo recipiente. Exceções são feitas às mensagens do tipo *broadcast* e *multicast*, envolvendo um número maior de processos recipientes. As rotinas de comunicação providas pela linguagem devem possuir um suporte confiável para troca de mensagens, seja este suporte prestado pelo sistema operacional da máquina ou implementado pela própria linguagem, no seu ambiente de suporte a execução.

O envio de uma mensagem se dá explicitamente por iniciativa do processo origem, enviando uma mensagem endereçada a um processo destinatário. A recepção por sua vez, pode ser tanto *implícita* como *explícita*. A recepção implícita de uma mensagem implica na criação automática de um novo fluxo de execução para tratar a mensagem recebida. No segundo caso, recepção explícita, o processo recipiente tem a possibilidade de selecionar mensagens, segundo algum critério pré-estabelecido, permitindo uma maior flexibilidade no tratamento. A forma

mais comum, e mais simples, é manter as mensagens ordenadas na ordem de chegada, tratando-as segundo a filosofia FIFO (*First In, First Out*) [BAL89].

As mensagens enviadas devem ser de alguma forma endereçadas aos destinatários, as formas mais comuns são o endereçamento *indireto* e o *direto*. O endereçamento indireto se dá através do envio da mensagem a um elemento intermediário, normalmente chamado de caixa-postal, o qual é acessado eventualmente pelo processo recipiente.

O endereçamento direto provê o envio de mensagens ao processo recipiente sem intermediários. Esta última forma de endereçamento pode ser *simétrica*, com ambos processos envolvidos conhecendo o endereço do parceiro, ou *assimétrica*, com apenas o processo origem conhecendo o endereço do processo destino.

Os modelos mais comuns implementados em linguagens para prover a comunicação entre processos são:

Mensagens assíncronas

Um processo origem **S** envia uma mensagem a um processo destinatário **R**, não aguardando sua recepção. As mensagens são mantidas em uma fila de espera em **R** até serem atendidas. Neste esquema, não há qualquer garantia de que o processo destinatário recebeu a mensagem nem há qualquer forma de sincronismo envolvido.

Mensagens síncronas

Um processo **S** envia uma mensagem a um processo **R** e permanece bloqueado, aguardando sua recepção, existindo ou não dados de retorno. Ao contrário do que pode ocorrer utilizando mensagens assíncronas, um processo **S** pode ter, no máximo, uma mensagem enviada em fila aguardando tratamento. Além de permitir a troca de informações, mensagens síncronas permitem garantir uma ordenação temporal das ações dos dois processos envolvidos na comunicação.

Rendezvous

É uma forma de comunicação síncrona, efetuada com um processo **S** invocando um procedimento especial em um processo **R**. A troca de dados é feita através de parâmetros enviados por **S** ao procedimento invocado em **R** e com o retorno de resultados. **S** executa em paralelo ao procedimento invocado em **R**.

RPC

RPC (chamada remota de procedimentos) possui sintaxe e semântica similar a uma chamada de procedimento local, permitindo envio de dados através de parâmetros a procedimentos remotos e de retorno de resultados. Difere do esquema *rendezvous* por bloquear o processo origem da invocação até o término do procedimento invocado.

Dispersão de mensagens

Os tipos de mensagens vistos acima permitem a comunicação entre dois processos: um origem e um destino. Variantes permitem a comunicação com todos os processos (*broadcast*) ou com um subgrupo (*multicast*). São as chamadas mensagens *um-para-muitos*.

2.4 Tolerância a Falhas

Em sistemas distribuídos existe a possibilidade de ocorrência de falhas parciais, onde um processador, ou um grupo, deixa de funcionar corretamente. Tais falhas devem ser detectadas e se possível corrigidas, não sendo esta uma tarefa simples. A responsabilidade por manter o sistema confiável deve ser dividida entre o sistema operacional, o ambiente de suporte à execução da linguagem e o programador [BAL89].

Nem todas as aplicações distribuídas necessitam suporte à detecção e correção a falhas parciais, resultando que, em alguns ambientes, a ocorrência de falhas parciais acarrete uma falha total da aplicação. O suporte mais simplista de tolerância a falhas parciais é deixar ao encargo do programador a tarefa de recuperação do erro, cabendo ao sistema operacional ou ao suporte de execução da linguagem a sinalização da ocorrência de algum evento não esperado.

Outras abordagens mais elaboradas são utilizadas, como transações atômicas e tolerância a falhas transparente à execução. Transações atômicas são bastante utilizadas ao serem manipulados dados em sistemas distribuídos. A principal característica das transações atômicas é a garantia da execução correta da operação. Caso não seja possível, a operação não é realizada.

Prover um ambiente com tolerância a falhas totalmente transparente durante a execução de uma aplicação é uma tarefa bastante complexa. O mecanismo de suporte a ocorrência de falhas parciais deve ser implementado sobre o sistema operacional ou sobre o ambiente de execução da aplicação, garantindo o funcionamento correto da aplicação. Estas estratégias, que diminuem as responsabilidades do programador, não são triviais, variando desde a replicação de processos à manutenção de arquivos de "log" com mensagens enviadas.

2.5 Sumário

Neste capítulo foram apresentadas definições de conceitos pertinentes a sistemas distribuídos, visando embasar a leitura do restante deste trabalho. Foi dada ênfase nas ferramentas e técnicas de suporte ao processamento distribuído utilizado nas linguagens de programação.

As definições apresentadas foram retiradas basicamente de [BAL89, HWA84], recorrendo-se a consultas ao material referenciado por estes autores.

3 LINGUAGENS ORIENTADAS A OBJETOS

O paradigma de orientação a objetos é utilizado em várias áreas da ciência da computação: projeto de sistemas, bancos de dados, linguagens e arquitetura de computadores, citando alguns exemplos encontrados na bibliografia. Apesar de vastamente utilizado, o conjunto de termos que definem os conceitos de orientação a objetos, não é de consenso entre os diversos autores, existindo diferentes interpretações sobre uma mesma característica [SNY93].

O paradigma de **programação orientada a objetos** [TAK90] é um exemplo da busca de simplicidade de escrita e compreensão de programas, oferecendo estruturas de encapsulamento de dados e favorecendo a escrita de grandes sistemas. Já os diversos modelos de **programação concorrente** [BAL89] visam apresentar recursos que possibilitem melhorar o desempenho de execução de aplicações, porém, por possuírem estruturas complexas, não são comumente utilizados em sistemas de grande porte.

A união dos paradigmas de programação orientada a objetos e concorrente provê os recursos desejados pelos projetistas de software, tanto no que tange ao desenvolvimento como no desempenho de execução de aplicações. Segundo [HOP89], há um crescente interesse no uso de técnicas orientadas a objetos como base para soluções concorrentes de uma vasta gama de problemas.

Este capítulo oferece uma visão geral do paradigma de programação orientada a objetos e tece considerações sobre suas implementações concorrentes. A abordagem segue a linha comumente adotada para linguagens de programação, utilizando como referências básicas os trabalhos de [TAK90] e [SNY93].

3.1 Programação Orientada a Objetos

Programas escritos segundo o paradigma de orientação a objetos são montados a partir de classes. As classes são estruturadas em uma hierarquia de generalização/especialização, modelando conceitos da aplicação. Quando em execução, objetos são instanciados a partir de classes. A interação entre os objetos, ou seja, a execução do programa, se dá através do envio de mensagens e do respectivo retorno de dados.

Segundo [TAK90], a tarefa de programação em uma linguagem orientada a objetos tem as seguintes etapas:

1. **concepção do modelo**: a aplicação deve ser modelada em objetos comunicando-se através de mensagens;
2. **construção de classes**: para cada grupo idêntico de objetos é abstraída uma classe para descrevê-los. Estas classes podem ser organizadas em níveis hierárquicos;
3. **descrever as classes**: programar na linguagem disponível as classes que podem herdar de uma possível "biblioteca de classes" propriedades, utilizando o recurso de generalização/especialização em níveis hierárquicos através do mecanismo de herança;
4. **composição do programa**: o início da execução do programa é a criação do primeiro objeto, caracterizando uma espécie de programa principal. Os demais objetos são criados direta ou indiretamente a partir desta primeira classe.

O estilo de programação segundo este paradigma introduz características bastante marcantes. A mais imediata delas é a **modularidade** encontrada no conceito de objeto que encapsula dados e funções. Este mesmo encapsula-

mento induz a uma **visão integrada entre dados** e **funções de acesso**, garantindo um nível de segurança na manipulação de informações e de autonomia de objetos.

Outras características estão fundamentalmente ligadas ao **suporte a estruturação** de conceitos em hierarquias de generalização/especialização, permitindo a **herança** de propriedades de uma classe a outra. Tais características incluem:

- composição de aplicações de forma *bottom-up*, partindo de classes já existentes;
- programação incremental e evolutiva, uma vez que pequenas modificações podem ser facilmente testadas e efetuadas, partindo da criação de novas classes ou alterações nas classes já existentes; e,
- reusabilidade de código de classes já escritas, depuradas e confiáveis, sob forma de biblioteca de classes.

3.1.1 Classes e herança

Um programa orientado a objetos consiste em um conjunto de *classes* que definem objetos de estrutura e comportamento idênticos. Esta característica é semelhante a um tipo de dados de uma linguagem convencional, como o tipo *integer* do Pascal que define variáveis com as propriedades do tipo inteiro, a classe define objetos com as propriedades com que foi construída. Um modelo genérico de classe contém os seguintes itens:

- nome da classe;
- lista de classes que transmitem propriedades a classe definida;

- declaração de variáveis que irão compor o estado interno de cada objeto desta classe; e,
- declaração de operações que irão compor os métodos que definirão o comportamento dos objetos desta classe.

Um quinto item contendo uma descrição informal do conteúdo também é bastante importante no que se refere à documentação da classe.

Uma classe pode ser construída a partir de outras já existentes, constantes da lista de classes, denominadas superclasses, que transmitem propriedades. A nova classe, denominada **subclasse**, define suas propriedades além de contar com as propriedades herdadas das superclasses. Uma subclasse pode também "editar" as propriedades de suas **superclasses**, especializando tarefas.

Normalmente, uma subclasse é uma especialização de suas superclasses, cujo enfoque é mais genérico. Uma propriedade genérica pode ser redefinida, especializando-a a solucionar um problema, que na implementação apresentada pela superclasse não é considerada a solução mais adequada. Havendo mais de uma definição para uma mesma propriedade, diz-se que esta é *sobrecarregada*[1].

A *herança* em linguagens orientadas a objetos é implementada de forma estática em tempo de compilação. Nas linguagens existentes há duas formas de herança: múltipla e simples. A herança simples implica que uma classe pode possuir uma única superclasse, enquanto que a herança múltipla permite existirem diversas superclasses para uma subclasse.

[1]Tradução do original em inglês *overloaded*.

3.1.2 Objetos

Objetos são instâncias de classes, de forma semelhante que em Pascal variáveis são instâncias de tipo de dados. Porém, ao contrário de variáveis comuns, os objetos são entes independentes, com capacidade de processamento, interagindo entre si através de mensagens, invocando ações em outros objetos. As propriedades de um objeto são as descritas pela sua classe mais as obtidas através de herança.

A figura 3.1 representa o modelo de um objeto genérico adotado pelas linguagens de programação orientadas a objetos. Os componentes desta representação estão descritos abaixo.

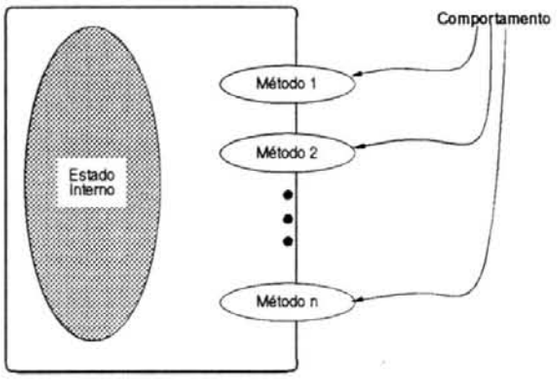

Figura 3.1: Modelo de um objeto

Estado interno , correspondente a memória interna ao objeto, registrando informações durante a vida do objeto. As variáveis mantidas nesta área de memória não são acessíveis senão através dos métodos do próprio objeto.

Comportamento do objeto , caracterizando as ações definidas pela classe que o objeto foi instanciado. O comportamento é composto por um conjunto de métodos, os quais são individualmente disparados pelo recebimento de uma mensagem.

Método , é equivalente a uma função, ou procedimento, fornecido pelo objeto. Um método corresponde a uma interface do objeto por onde é permitido invocar uma determinada ação.

Durante a execução de um programa, os objetos não são criados espontaneamente ou ao acaso, mas sim através de outro objeto. A remoção de um objeto também é explícita. A duração da "vida" de um objeto pode ser efêmera, durante a execução de um programa, ou persistente, com tempo de vida indefinido, contando sua implementação com a possibilidade de retê-lo em uma base secundária de dados.

Os objetos são distinguidos entre si através de um identificador, o qual é único e acompanha o objeto durante toda a sua vida. Nas linguagens de programação convencionais, a forma mais comum de identificação de um objeto é através de sua posição de memória.

3.1.3 Comunicação entre objetos

Como dito anteriormente, objetos interagem entre si através de mensagens. Estas mensagens são compostas basicamente por dois campos: um **seletor** da operação desejada no objeto recipiente e da **lista de parâmetros** necessários a sua execução. Como resposta ao recebimento de uma mensagem, o objeto recipiente ativa o método especificado pelo campo de seleção podendo, ao final de sua execução, retornar alguma informação.

Para linguagens orientadas a objetos é essencial que um único seletor possa identificar diversas operações distintas, ou seja, um seletor pode, dependendo da classe a que pertence o objeto recipiente, ativar diferentes métodos. Já ALGOL-68 havia implementado esta característica de **sobrecarga de operadores**, por exemplo, o operador "+" (adição) pode ser utilizado para selecionar

uma adição inteira ou real, concatenar *strings* ou ainda para realizar uma outra operação definida pelo programador.

O correto funcionamento da sobrecarga de operadores somente é garantido se o acoplamento mensagem/método ocorrer dinamicamente em tempo de execução e não de forma estática, no momento da compilação do programa.

Também o *polimorfismo* é suportado nas linguagens orientadas a objetos como forma de caracterizar sobrecarga de seletores. Basicamente, o polimorfismo consiste em diferenciar dois métodos com o mesmo seletor. Em C++, o polimorfismo é implementado em tempo de compilação, diferenciando os métodos através de seus parâmetros.

3.2 Implementações Distribuídas

A definição de sistemas de computação distribuídos de [BAL89] compreende o comportamento de componentes fisicamente independentes e de módulos com autonomia lógica comunicando-se por meio de troca de mensagens. Outra definição para sistemas distribuídos foi apresentada em [WEG92]: um sistema cujos componentes possuem limites de encapsulamento, permitindo acessos aos recursos dos componentes somente por meio de uma rígida interface de comunicação, composta por um protocolo de acesso aos módulos.

Esta definição deixa claro que sistemas distribuídos provêem recursos físicos para implementação de sistemas orientados a objetos, uma vez que estes possuem características similares (tabela 3.1). Nesta relação, a independência a nível de execução de módulos de sistemas distribuídos podem ser providos por objetos encapsulando dados e funções. A abstração de dados provida pelo paradigma de orientação a objetos provê os recursos para implementação do protocolo de acesso aos serviços dos módulos distribuídos. As mensagens entre

os módulos de um sistema distribuído são mapeadas em um ambiente de objetos através de invocações de métodos.

Tabela 3.1: Sistemas distribuídos *vs.* orientação a objetos

Orientação a Objetos	**Sistemas Distribuídos**
encapsulamento de objetos	limites de execução de um módulo
estado interno	memória interna
métodos	serviços prestados
abstração de dados	protocolo de acesso
invocação de métodos	mensagens entre módulos

Esta seção analisa os pontos a serem considerados ao implementar de forma distribuída uma linguagem orientada a objetos. Os pontos a serem considerados são:

- o nível de concorrência obtido;
- os processos de comunicação e sincronização entre objetos;
- a estrutura de implementação do objeto;
- a gerência do ambiente; e,
- a implementação de mecanismo de herança.

3.2.1 Nível de concorrência

A abordagem da concorrência em linguagens orientadas a objetos pode ser realizada em dois níveis distintos:

inter-objetos , com objetos executando de forma concorrente entre si; ou,

intra-objeto , com a execução concorrente de diferentes fluxos de instrução de um mesmo objeto.

O modelo de concorrência **inter-objetos** implica que um objeto possa tratar apenas uma mensagem em um determinado instante de tempo, permitindo a concorrência somente a nível de objetos. Possui como característica a simplicidade de modelar um objeto, havendo a necessidade de apenas dois canais de comunicação: um canal de entrada para invocações de serviços e outro para efetuar seus envios de mensagens.

A modelagem da concorrência **intra-objeto** prevê sua implementação conjunta ao modelo de concorrência inter-objeto. Neste esquema, vários fluxos de controle internos a um objeto podem estar executando de forma concorrente. Neste caso, a modelagem do objeto deve conter algum mecanismo de controle de acesso ao estado interno, como uso de semáforos ou monitores provendo exclusão mútua.

O objeto com a capacidade de concorrência interna deve possuir tantos canais de entrada de mensagem quantos forem os números de métodos que possam ser executados de forma concorrente, ou possuir um único canal, manipulado por um processo gerenciador, o qual decodifica a mensagem recebida criando um novo fluxo de execução (*thread*) para tratá-la.

A concorrência inter-objetos já é garantida pelo modelo de orientação a objetos, cabendo aumentar seu grau através da introdução do esquema de concorrência intra-objeto e/ou por pelo menos um dos esquemas citados na seção 2.2.1, discutidos abaixo.

3.2.1.1 Modelo de objeto ativo

O modelo de objetos prevê três estados para os objetos: repouso, processando ou aguardando. Um objeto encontra-se em repouso quando não estiver processando uma mensagem recebida. O estado em que encontra-se processando é o em que está realizando um serviço solicitado. O objeto encontra-

se aguardando quando enviou uma solicitação de serviço a outro objeto e está a espera de uma resposta. Neste esquema, o modelo do objeto é dito *passivo*.

Permitir que o objeto processe algum serviço sem ter recebido nenhuma mensagem, ou prossiga a execução após enviar resposta a uma mensagem recebida, implica na adoção do modelo de objeto *ativo*. Neste caso, a concorrência é garantida por vários objetos executando suas tarefa em um mesmo instante de tempo, independente do envio ou não de mensagens. É previsto a possibilidade de uma mensagem recebida interromper a execução normal interna a um objeto.

3.2.1.2 Troca de mensagens

O modelo natural de comunicação entre objetos é realizado através de mensagens síncronas: um objeto **A** invoca um método do objeto **B**; o objeto **A** aguarda a resposta do objeto **B** ao final do processamento do método invocado. Este modelo implica que, em um determinado instante, apenas um objeto esteja processando, pois pelo paradigma de orientação a objeto não há concorrência intra-objeto, havendo apenas um fluxo de execução por objeto.

Um maior número de objetos processando pode ser obtido através da utilização do modelo de concorrência intra-objeto, o que implicaria no bloqueio de apenas uma das *threads* do objeto; outra é a utilização de modelos alternativos de comunicação: mensagens *assíncronas* e *múltiplas*.

A comunicação assíncrona permite que um objeto **A** envie uma mensagem a um objeto **B** sem aguardar o término do processamento do método invocado, implicando na necessidade de não existirem dados com retorno de informações de **B** para **A**. Desta forma, **A** e **B** executam de forma concorrente suas tarefas. Algumas linguagens permitem o uso de mensagens assíncronas contando com retorno de resultados. Estas linguagens utilizam-se de estruturas

auxiliares como, em ABCL/1 [YON86], variáveis futuras, e, em ConcurrentSmalltalk [YOK86], caixas-postais.

Outra forma de aumentar o grau de concorrência entre objetos é prover mecanismos de dispersão, possibilitando o envio de mensagem a um grupo de objetos (*multicast*) ou a todos os objetos (*broadcast*), permitindo o disparo de ações concorrentes em vários objetos.

3.2.2 Sincronização

A sincronização em uma linguagem concorrente orientada a objeto deve ser considerada sobre dois aspectos. O primeiro relacionado à sincronização entre os objetos e o segundo, no caso de existir a concorrência intra-objeto, à sincronização interna de um objeto.

Entre objetos o sincronismo é efetuado pelo mecanismo de comunicação provido, que pode ser implementado de forma **implícita** ou **explícita**. A forma implícita implica que o programador não utilize diretamente estruturas de manipulação de mensagens, mas que estas estejam embutidas na própria manipulação do objeto. A recepção neste caso pode controlar o recebimento de mensagens através de uma fila simples, como em POOL-T [AME87], ou atribuindo níveis de prioridades, como em ConcurrentSmalltalk e ABCL/1.

Esquemas com tratamento explícito de mensagens são mais flexíveis, podendo o programador incluir pontos de sincronismo a qualquer momento da execução, permitindo também implementar diferentes políticas de prioridades na recepção. Como custo ao maior grau de flexibilidade disponível, cabe ao programador manipular as tarefas de envio e recebimento de mensagens, tornando a programação mais complexa. Argus e Eiffel [MEY987] são linguagens com tratamento explícito de mensagens na comunicação entre objetos.

A nível interno ao objeto, a sincronização é necessária quando há o compartilhamento no acesso ao estado interno do objeto. Nestes casos, é comum o uso de mecanismos de acesso a seções críticas, como é o caso das implementações de Argus e ConcurrentSmalltalk.

3.2.3 Compartilhamento de código

Herança e delegação são mecanismos que permitem estender as propriedades dos objetos. A herança permite a especialização de classes, contando com o reuso de código em uma hierarquia de classes. Delegação é baseada na idéia de um objeto repassar a outro uma mensagem que não é capaz de resolver. A delegação é utilizada em linguagens concorrentes segundo o modelo de atores [AGH86], ABCL/1. Herança é implementada em linguagens orientadas a objetos, como Hybrid [NIE87].

Em linguagens que implementam delegação, o compartilhamento de código é feito dinamicamente em uma hierarquia de instâncias de classes, ou seja, em tempo de execução, entre objetos. O mecanismo consiste em um objeto "delegar" a responsabilidade de realizar uma operação a outro objeto quando, ao receber uma mensagem, não puder resolvê-la.

A herança pode ser vista como uma especialização do modelo de delegação [WEG87], onde existe o compartilhamento de código entre as classes. Uma classe, denominada *subclasse*, pode ser definida como a especialização de outra classe, *superclasse*, herdando desta suas propriedades, podendo alterá-las e incluir suas próprias.

A implementação do mecanismo de herança pode ser *estático*, com o compilador copiando o código da superclasse para a subclasse, ou *dinâmica*, onde, em tempo de execução, o código que implementa as classes é compartilhado pela

hierarquia de herança. A herança dinâmica é especialmente útil quando existe o compartilhamento de métodos, não havendo a necessidade da replicação de código.

Enquanto que, utilizando herança estática, o gasto de memória é considerável, a herança dinâmica não possui desempenho de execução tão eficiente, além de contar com vários problemas de sincronismo entre métodos concorrentes [WYA92]. A maioria das linguagens que implementam o recurso de herança estática utilizam o mecanismo de cópia, como em Argus e Eiffel.

Algumas linguagens para evitar os problemas advindos do uso de herança em ambientes concorrentes optam por não implementar este recurso, violando uma das características principais das linguagens orientadas a objetos [WYA92], e reduzindo o potencial da linguagem. POOL-T e Presto [BER88] optaram por não implementar herança.

3.2.4 Gerência do ambiente

O controle em tempo de execução de uma linguagem orientada a objetos concorrente deve prover mecanismos a nível de gerência de processos, de localização de objetos e mecanismos de tolerância a falhas. Tais controles, quando realizados automaticamente, pelo ambiente de execução, simplificam a tarefa do programador, além de garantir um melhor uso dos recursos.

3.2.4.1 Processos

Objetos são alocados a processos de forma a terem suporte para execução, contando com capacidade de processamento e memória. A criação de processos pode ser *estática* com um número fixo de processos definidos e cria-

dos no início da execução do programa, como implementado em Parmacs, ou *dinâmica*. Com uma maior flexibilidade para o programador, a criação dinâmica de processos pode ser implementada de forma *implícita* ou *explícita*.

Nas linguagens com criação explícita, são providos mecanismos de criação de processos externos ao objeto. Neste modelo, a concorrência está integrada na estrutura do objeto. Internamente ao objeto, a integridade é mantida por meios explícitos de controle, como semáforos e monitores. No modelo implícito, processos são internos a objetos. O recebimento de uma mensagem por um objeto pode ocasionar múltiplos fluxos de execução internos.

Tal como em ABCL/1 e ConcurrentSmalltalk, a maioria das linguagens implementam a criação implícita de processos, simplificando a tarefa de programação, uma vez que não é necessária a especificação das atividades paralelas. Argus e Presto são linguagens que implementam mecanismos de criação explícitos.

Também *implícita* ou *explícita* são as formas adotadas para o término de processos. A forma implícita implica em que um processo termina ao enviar resposta a uma mensagem recebida. Este modelo possui como característica um custo maior em tempo de execução, devido a constante criação e remoção de processos que atendem a uma única invocação. Muitas linguagens baseadas no modelos de Atores[2] utilizam este mecanismo, com é o caso de Actor.

A remoção explícita de processos, implementada em ABCL/1 e ConcurrentSmalltalk, permite que um objeto atenda a diversas mensagens, até receber uma que comunique seu término.

Um processo encontra-se ativo enquanto estiver executando uma tarefa. Processos são implementados de forma a permanecer ativos mesmo sem ter recebido mensagens ou sendo ativados no momento em que receberem uma

[2]No modelo de Atores, um objeto responde somente a uma mensagem.

invocação. Com processos que permanecem ativos, o grau de concorrência é maior, porém pode ocorrer gastos de recursos desnecessários, no caso de algum processo não ter qualquer tarefa a realizar. O segundo caso, onde processos são ativados por mensagens, há um custo adicional devido a ativação de processos "dormentes", porém, por ser mais próxima ao paradigma de orientação a objetos, a modelagem e a programação é mais simples [WYA92].

3.2.4.2 Localização

Uma vez criados, os objetos concorrentes devem possuir um identificador único que diferencie um objeto dos demais, sendo possível o endereçamento de mensagens. Em ambientes seqüenciais ou em ambientes concorrentes com memória única é possível utilizar o endereço de memória como identificador de um objeto. Em ambientes distribuídos, é necessário conhecer o nodo que está processando o objeto e o canal de comunicação do objeto destino de uma mensagem.

As linguagens distribuídas, na medida do possível, devem prover transparência de localização de objetos. Em outras palavras, no acesso a objetos distribuídos, a mesma semântica utilizada para acessar objetos locais deve também prover acesso a objetos remotos [WEG92]. Desta forma a tarefa de programação é bastante simplificada, uma vez que a referência a objetos é feita de forma independente de sua localização.

Em sistemas distribuídos processos podem estar interconectados de forma *estática*, com a conexão de cada processo definida no momento da criação, ou *dinâmica*, podendo as ligações serem estabelecidas e alteradas durante a execução do processo. Este último esquema possibilita uma maior flexibilidade na manipulação dos objetos, onde os canais de comunicação (portas) consistem em variáveis, as quais podem ser associadas conexões.

Existindo a possibilidade de criação dinâmica de objetos (processos), o ambiente de execução deve utilizar um *servidor de nomes* que garanta a não ocorrência de identificadores iguais. Também deve ser garantida a integridade do identificador se for permitida a migração de objetos.

Dois esquemas são adotados para localizar objetos em um ambiente distribuído. O primeiro, totalmente transparente, deixa ao encargo do ambiente de execução a tarefa de "descobrir" o endereço de um objeto a partir de seu identificador. Para isto, o ambiente deve contar com um servidor de nomes (centralizado ou distribuído). No segundo esquema, o endereço físico do objeto está embutido no seu identificador. Esta segunda forma produz uma execução mais eficiente, porém limita o potencial de migração de objetos (esquema utilizado em Argus). O modelo proposto por [ROS91] utiliza servidores de nomes distribuídos.

3.2.4.3 Tolerância a falhas

A implementação de mecanismos de tolerância a falhas é fortemente desejável em linguagens para sistemas distribuídos, porém nem sempre é uma tarefa trivial. Muitas propostas de linguagens não abordam este item devido à sua complexidade.

Algumas linguagens apresentam recursos para suportá-la utilizando operações atômicas sobre objetos atômicos, sendo este o caso de Argus. Outros mecanismos utilizados são baseados em *recuperação*, que, em caso de falha retorna ao último estado consistente conhecido, ou em *replicação* de objetos.

3.2.4.4 Processadores

Em linguagens para ambientes distribuídos, um objeto ao ser instanciado, é associado a um processo criado em algum nodo processador. Segundo

[CHI91], o desempenho ótimo de uma aplicação é obtido distribuindo os objetos que a compõem em um grupo de processadores próximos e com baixa carga de processamento. Desta forma é garantido um maior potencial de processamento e a redução do tempo total de comunicação.

A escolha do nodo para processamento do objeto pode ser *automática*, provida pelo ambiente, ou *manual*, por interferência direta do programador. Sendo implementada por mecanismos automáticos, o ambiente de execução seleciona o nodo com menor carga de processamento para o objeto criado, supondo desta forma prover melhor desempenho geral da aplicação.

Métodos manuais contam com comandos, inseridos na linguagem, permitindo a especificação de um determinado nodo para processamento do objeto. Através desta forma de alocação de objetos a processadores espera-se que o programador opte por nodos próximos para alojar objetos que necessitem comunicação freqüente, melhorando desta forma o desempenho final com a redução do tráfego de mensagens.

A alocação manual permite também alocar objetos que utilizem recursos específicos de determinados nodos de processamento que possam suprir estas necessidades diferenciadas. Por exemplo, em uma rede de estações de trabalho, objetos que utilizam saídas gráficas podem ser alocados em nodos com vídeo gráfico e objetos que realizam operações de entrada e saída em arquivos, em nodos com unidades de disco.

Outro mecanismo utilizado para aumentar o desempenho em linguagens distribuídas é permitir a migração de processos. Em ambientes orientados a objetos, este esquema consiste em retirar um objeto processando em uma máquina sobrecarregada alocando-o em outra, com capacidade de processamento ociosa. Porém nem sempre é vantajosa a implementação de migração de objetos [CHI91], em função da necessidade de bloquear o objeto, redirecionar as mensagens rece-

bidas durante o processo de migração, além do tempo gasto no processamento de cópia propriamente dito.

3.3 Sumário

Este capítulo abordou o paradigma de orientação a objetos segundo os conceitos comumente utilizados nos trabalhos relativos às linguagens de programação. Em especial foi dada ênfase na caracterização de linguagens orientadas a objetos concorrentes, abordando questões de sincronismo, gerência de ambiente e implementação de herança, entre outros tópicos.

A leitura deste capítulo auxilia na interpretação das necessidades de um modelo distribuído para linguagens orientadas a objetos concorrentes.

4 O MODELO DISTRIBUíDO

Este capítulo apresenta uma proposta de um modelo para suportar a execução distribuída de objetos, unindo as características do processamento distribuído com a programação e desenvolvimento de software segundo o paradigma de orientação a objetos. O resultado obtido pelo emprego do modelo é a obtenção de uma linguagem orientada a objetos com capacidade de processamento concorrente.

O modelo de objetos distribuídos apresentado pode ser visto de duas formas distintas [CAV93b]: a visão a nível de usuário e a visão a nível de operação do modelo. A nível de usuário, os mecanismos utilizados para implementação dos recursos de distribuição são transparentes, sendo a atenção totalmente dirigida para a linguagem de programação de suporte ao modelo. O nível operacional, por sua vez, provê os meios que garantem a execução distribuída de objetos de um programa.

O nível *operacional* do modelo é totalmente independente da linguagem que o implementa, assim como a sua operação é transparente para o usuário. Esta transparência é provida por mecanismo de compilação, responsável por introduzir no programa o modelo operacional de objetos distribuídos. O nível de *linguagem* consiste na ferramenta utilizada para a programação de sistemas distribuídos.

A figura 4.1 mostra como se dá a união dos diferentes níveis do modelo proposto através de uma compilação de características. O processo de compilação une um programa escrito em uma linguagem que obedece as características do nível de linguagem ao modelo de execução do nível operacional. Como resultado, é obtido um ambiente distribuído de execução de uma aplicação.

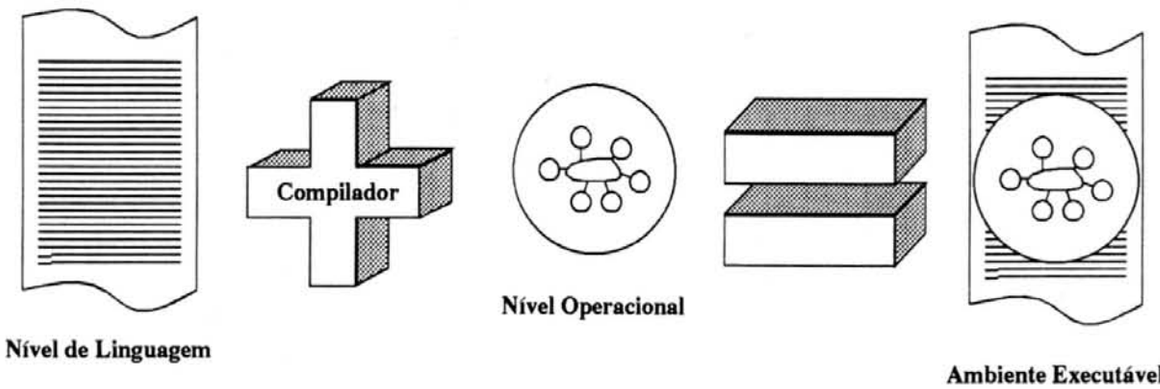

Figura 4.1: Geração do ambiente de execução distribuído

4.1 Modelos da Literatura

Na literatura são encontrados diversos modelos de linguagens orientadas a objetos que exploram sistemas concorrentes, paralelos ou distribuídos. Diferentes soluções são utilizadas para implementar as características necessárias para um sistema distribuído segundo a meta buscada no desenvolvimento da linguagem. Apesar disto não se pode dizer que uma solução é melhor que outra, apenas que a opção por utilizar uma determinada solução apresenta benefícios em para a utilização ao qual a linguagem é voltada.

O modelo proposto objetiva a simplicidade de escrita de programas em ambientes distribuídos. Assim, programadores com pouca ou nenhuma experiência em ambientes distribuídos tem acesso a um ambiente de programação de melhor desempenho. A definição do presente modelo partiu da experiência relatada na bibliografia de outros modelos.

Para manter as características de uma linguagem orientada a objetos, o modelo implementa a opção de herança através de cópia de código. Igual solução adotado por Argus. Utilizar o mecanismo de delegação, do modelo de atores [OKA94], distanciaria o modelo do usuário comum.

O paralelismo é incorporado de forma explícita, podendo o usuário definir a granulosidade de execução que a aplicação irá ter. Porém, o número de processos da aplicação é variável, não havendo a necessidade do programador criar ou destruir processo explicitamente.

O paralelismo no modelo é explorado através do envio de mensagens assíncronas entre objetos. Esquemas de sincronismo podem ser obtido com outros dois tipos de mensagens síncronas: mensagens em que há fluxo bidirecional de dados e mensagens onde é sinalizado o início do tratamento de uma solicitação.

A transparência de localização de objetos é garantido pelo mecanismo de endereçamento implícito. No modelo, o mecanismo de endereçamento é garantido pelo uso de elementos com funcionalidade semelhante às funções *stubs* de RPC e à solução adotada pelo modelo de computação reflexiva: um elemento de processamento separado do espaço de execução do programa garante o mecanismo de transferência de mensagens.

A tabela 4.1 apresenta um quadro com o resumo das características de implementação do modelo proposto. No capítulo 2.5 são feitas referências às soluções encontradas para implementações de outros modelos de linguagens orientadas a objetos concorrentes.

Tabela 4.1: Características de implementação do modelo proposto

Característica	**Implementação**
Incorporação do paralelismo	Explícito
Herança	Por cópia
Número de processadores	Variável
Exploração da concorrência	Mensagens assíncronas
Comunicação	Direta com filas FIFO
Endereçamento	Implícito
Localização	Procuradores

4.2 O Modelo Operacional

O modelo básico de objetos distribuídos é visto na figura 4.2, contando com n objetos executando de forma concorrente entre si e cooperando através de uma malha de comunicação. Cada objeto possui sua própria área de memória para objetos locais, bem como possui fluxo de execução próprio independente dos demais. A comunicação entre objetos é efetuada através de mecanismos de troca de mensagens pela rede de comunicação.

Figura 4.2: Modelo básico de objetos distribuídos

4.2.1 Suporte de hardware

A estrutura ideal de hardware para o modelo da figura 4.2 consiste em uma arquitetura que possua tantos nodos processadores, com memória própria, quantos forem os objetos a serem executados em uma determinada aplicação. A malha de interconexão, por sua vez, deve permitir a comunicação simultânea entre pares formados arbitrariamente entre todos os processadores.

As figuras 4.3 e 4.4 apresentam uma estrutura ideal de hardware para suporte à execução do modelo. A figura 4.3 mostra um nodo de processamento do modelo. Este nodo conta com uma unidade de processamento para execução do código do objeto (UCP - Unidade Central Processamento) e de um módulo de

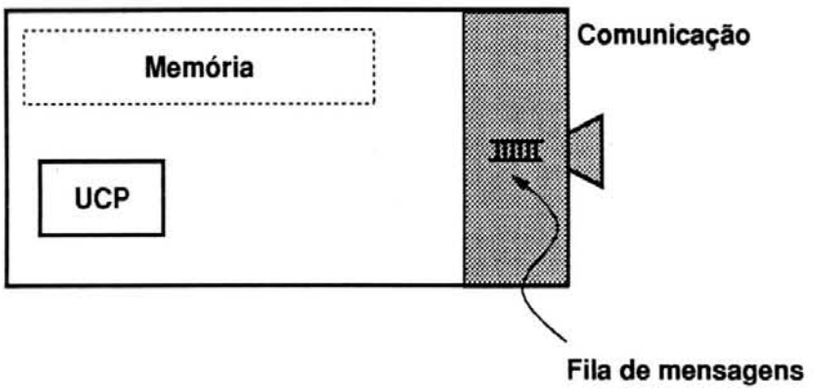

Figura 4.3: Nodo de processamento

memória para suporte aos objetos locais e do código que implementa o objeto. O nodo também dispõe de uma interface de comunicação que possibilita tanto o envio como o recebimento de mensagens. Esta interface mantem as mensagens recebidas em uma fila, segundo a filosofia FIFO, aguardando o seu tratamento pelo objeto.

A interconexão ideal para prover a comunicação entre os nodos é apresentada pela figura 4.4. Nesta configuração todos os nodos possuem uma ligação direta com os n nodos que compõem a malha de processadores disponíveis. Com este tipo de ligação é possível até $n/2$ conexões simultâneas, com tempo mínimo para comunicação entre objetos, por dispor de um canal dedicado para cada conexão.

A dificuldade de precisar o número de objetos que irão compor uma aplicação e o alto custo que teria uma arquitetura com tantos nodos processadores quantos fossem necessários a qualquer aplicação, ou seja, número infinito de nodos, inviabilizam a construção de uma máquina ideal para suporte ao modelo. Possibilitando que um nodo processador seja compartilhado entre diversos objetos, viabiliza-se a construção de uma arquitetura para suporte ao modelo. Em [BRO86] é apresentada a adequação em uma plataforma real de um modelo ideal de arquitetura de suporte a uma linguagem orientada a objetos.

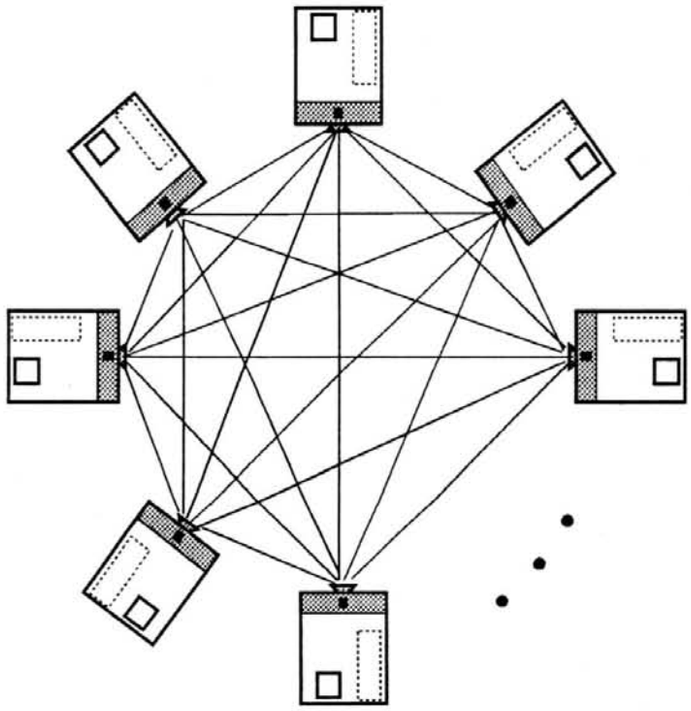

Figura 4.4: Implementação ideal da arquitetura de suporte ao modelo

Sendo de implementação bastante simples no modelo ideal, o nodo processador que possibilite o compartilhamento de seus recursos entre diversos objetos necessita de uma implementação mais robusta. A primeira modificação no modelo do hardware ideal consiste em introduzir uma memória com maior capacidade de armazenamento. Esta memória deve ser segmentada, sendo que cada segmento é ocupado por um objeto. Estes segmentos são por sua vez divididos em duas partes, uma contando com o código executável do objeto e a outra com a memória privada (para objetos locais). Quando em execução, o objeto tem acesso somente ao segmento a ele pertinente.

Com o compartilhamento da unidade de processamento, surge a necessidade de um **escalonador** de objetos. A tarefa deste escalonador consiste em viabilizar o uso da UCP entre os diversos objetos. O escalonador retira a porção executável de um objeto da memória, alocando a UCP para seu uso por um tempo fixo determinado. Tendo este tempo terminado, o objeto retorna à memória, sendo a UCP liberada para outro objeto. A interface de comunicação deve ter sua funcionalidade aumentada a fim de suportar o gerenciamento de

mensagens a diversos objetos. Para tanto, deve manter uma fila de mensagens distinta para cada objeto em execução no nodo.

Estas alterações ao modelo do nodo de processamento ideal são visualizadas na figura 4.5. O módulo de memória encontra-se ocupado com diversos objetos, sendo que um destes ocupa a UCP.

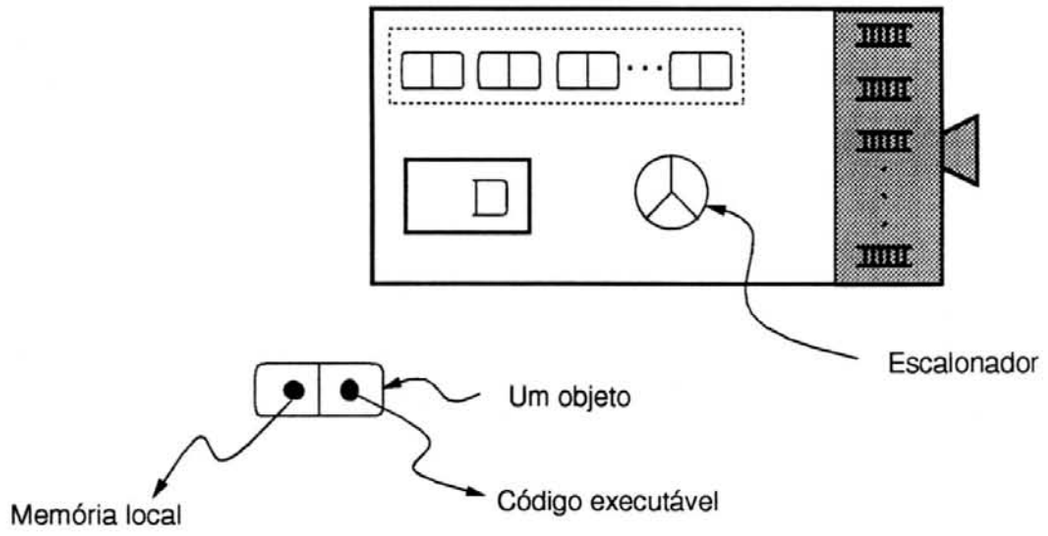

Figura 4.5: Nodo de processamento de suporte à execução

Com um número de nodos de processamento definido, é viável a construção de uma malha conectando todos os nodos processadores. Diferentes topologias de interconexão podem ser utilizadas para este fim como mostra a figura 4.6. Estas diversas topologias podem ser analisadas segundo suas características, por exemplo, uma interconexão do tipo barramento possui um custo de implementação bastante baixo, enquanto que interconexões do tipo *crossbar* possibilitam um melhor desempenho.

Em relação ao modelo de objetos distribuídos proposto, a arquitetura de interconexão é irrelevante, conquanto o elemento responsável pela comunicação nos nodos processadores seja configurado para processar a troca de mensagens corretamente.

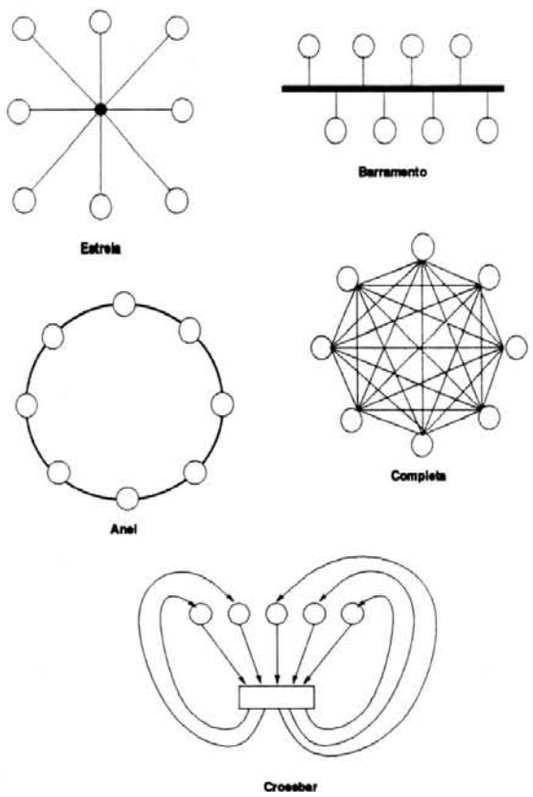

Figura 4.6: Interconexões de nodos processadores

Considerando as restrições apresentadas para configurar o hardware de suporte, o modelo trabalhado é apresentado na figura 4.7. Nesta representação tem-se um número fixo p de nodos de processamento, cada um suportando a execução de até d objetos, onde d é determinado pela capacidade de memória dos nodos. A interconexão entre os nodos pode ser organizada como qualquer uma das topologias apresentadas na figura 4.6, sendo neste caso, o elemento de comunicação do nodo de processamento capaz de suportá-la.

4.2.2 O objeto distribuído

O objeto distribuído consiste na unidade básica de execução do modelo apresentado, sendo composto por três elementos, como mostra a figura 4.8:

- interface de acesso;

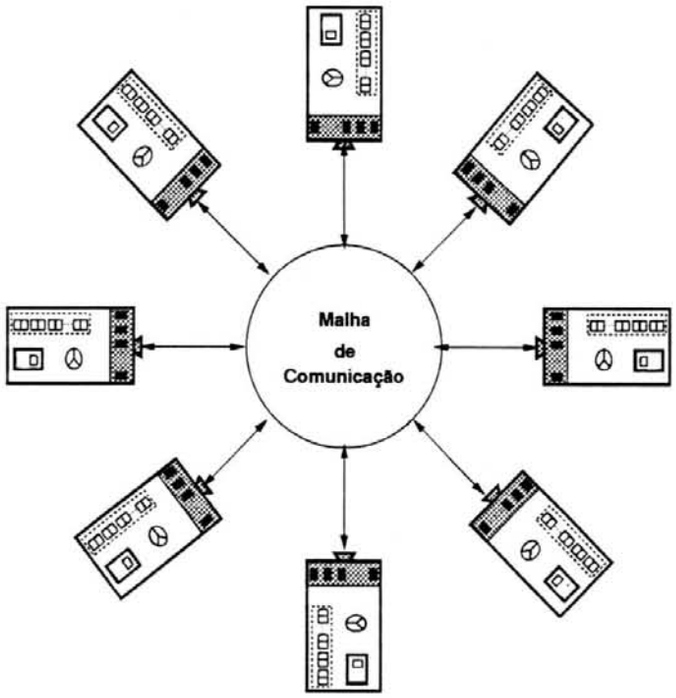

Figura 4.7: Arquitetura de suporte a execução do modelo

- implementação dos serviços; e,
- memória para objetos locais.

4.2.2.1 Interface de acesso

A interface de acesso possui a tarefa de receber as invocações de métodos do objeto distribuído ativando o método correspondente. Outra tarefa consiste em realizar o retorno de resultado, quando este serviço se fizer necessário. Os elementos componentes da interface de acesso são o servidor de mensagens, responsável pela comunicação e o elemento de delegação, responsável pela ativação dos métodos do objeto.

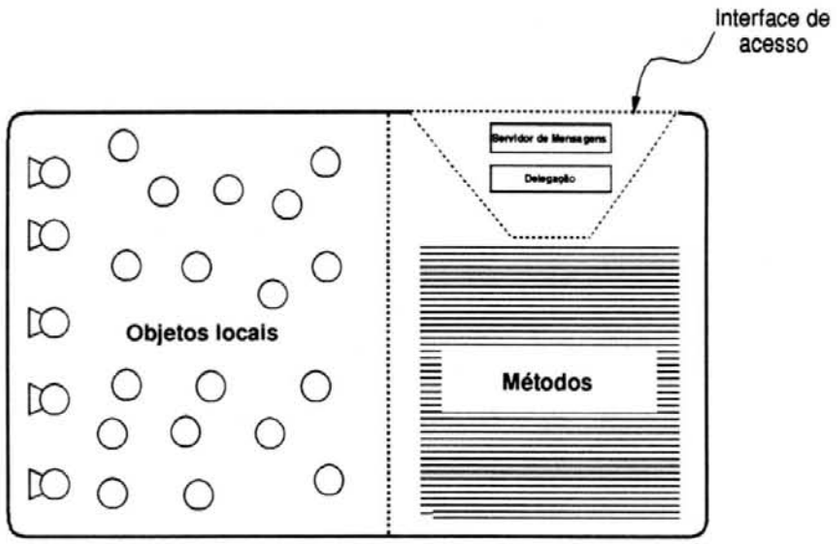

Figura 4.8: Modelo do objeto distribuído

4.2.2.2 Servidor de mensagens

O servidor de mensagens do objeto está ligado a uma das filas de mensagens providas pelo elemento responsável pela comunicação do nodo processador. Este servidor provê um canal de entrada para invocações de métodos do objeto, bem como um mecanismo para envio de resultados das operações solicitadas, permitindo que o mundo externo se comunique com o objeto.

As mensagens recebidas contém dois campos, a identificação de **endereço** do objeto originador da mensagem e um campo de **informação**. Este último é composto pela discriminação do método solicitado e dos parâmetros que este método necessita. O identificador de endereço do objeto origem permite o envio de mensagens de retorno de resultados, sendo composto pelo nome do nodo onde está executando o objeto origem e o código da "caixa-postal" para onde deve ser enviada a mensagem de resposta.

Enquanto o campo de informação é repassado ao elemento de Delegação, descrito abaixo, o campo de identificação do endereço do objeto origem é mantido pelo servidor de mensagens.

4.2.2.3 Delegação

O serviço prestado pelo elemento de delegação consiste em ativar a tarefa solicitada pela mensagem recebida, procedendo o envio de parâmetros ao método correspondente. No caso de ser constatada a necessidade de retorno de resultados ao objeto invocador, cabe ao elemento de delegação requisitar ao servidor de mensagens o envio de uma mensagem de resposta, bem como fornecer os dados a serem enviados.

Ao receber uma mensagem invocando algum método, o elemento de delegação bloqueia o recebimento de mensagens até o término da execução do método ativado. O servidor de mensagens é liberado a receber uma nova mensagem através de uma sinalização do elemento de delegação, o qual detém o controle do final da execução dos métodos ativados.

4.2.3 Serviços oferecidos

Os serviços oferecidos pelo objeto consistem nos métodos que podem ser invocados pelos demais objetos. Estes métodos podem ou não receber parâmetros e retornar resultados. Os métodos são executados de forma seqüencial, ativados pelo elemento de delegação, podendo acessar os objetos locais. Uma vez terminada a execução, o método retorna o controle para o elemento de delegação.

Considerando os tipos de retorno de resultados possíveis, os métodos são divididos em três categorias: sem retorno, com confirmação e com retorno. A invocação de um método sem retorno implica na sua execução sem qualquer sinalização ao objeto que realizou a chamada. O controle do momento de início da execução de uma tarefa solicitada pode ser conhecido através da utilização

de métodos com confirmação. O elemento de delegação, ao ativar um método que confirma seu recebimento, envia uma mensagem ao objeto originador sinalizando que iniciará o tratamento da mensagem recebida. Métodos com retorno de resultado possibilitam que, ao final de sua execução, sejam retornados dados para o objeto que solicitou o serviço.

Diferentes níveis de sincronismos são obtidos através da utilização dos diferentes tipos de métodos. A interação utilizando métodos sem retorno se dá de uma forma totalmente assíncrona, enquanto que invocações a métodos com retorno de resultados são síncronas. Em uma posição intermediária, métodos com confirmação permitem que o objeto originador conheça o momento do início do tratamento de sua solicitação, porém não do seu término.

4.2.4 Objetos locais

Objetos locais consistem em objetos manipuláveis somente na área de endereçamento do objeto no qual está inserido. Esta área consiste no estado interno do objeto. Estes objetos são criados e destruídos conforme a necessidade da execução dos métodos ativados. São dois os tipos de objetos locais: **objetos de dados** e **objetos procuradores**.

4.2.4.1 Objetos de dados locais

Objetos de dados locais consistem em objetos cujas informações são pertinentes unicamente ao objeto que os instanciou. Caracteristicamente possuem métodos que não demandam excessivo tempo de processamento, não compensando o *overhead* de executá-lo de forma distribuída. A figura 4.9 mostra os componentes de um objeto de dados local, dados contendo informações e métodos de acesso a estas informações.

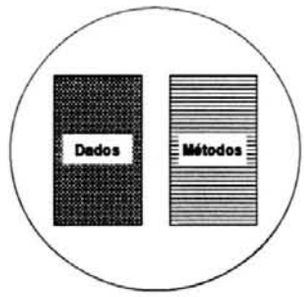

Figura 4.9: Modelo de objeto local

4.2.4.2 Objetos procuradores

Os objetos procuradores são os elementos do modelo que permitem o envio de mensagens a objetos que estejam em outra área de endereçamento, bem como receber respostas das solicitações realizadas. Cada objeto procurador representa um único objeto distribuído, consistindo em uma imagem deste. A figura 4.10 apresenta o esquema de um objeto procurador, cujos componentes são discutidos abaixo.

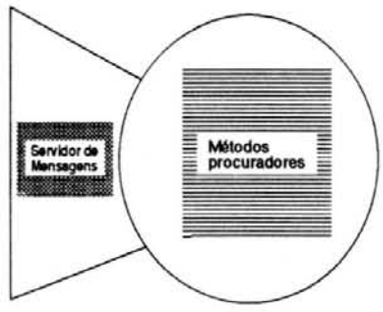

Figura 4.10: Modelo de objeto procurador

Servidor de mensagens : responsável pelo envio de mensagens de invocação e do recebimento de respostas às mensagens enviadas. Este servidor está associado a uma fila de mensagens, controlada pelo servidor de comunicação do nodo, a qual é utilizada como **caixa-postal** para recebimento de respostas às invocações realizadas. O servidor de mensagens também possui a identificação do objeto que representa, possibilitando enviar mensagens de invocação.

Métodos procuradores : cada método do objeto distribuído possui como correspondente um método procurador no objeto procurador. Este método procurador possui o mesmo protocolo de acesso, ou seja, recebe os mesmos parâmetros e retorna os mesmos resultados que o método que representa. Porém não implementa a tarefa propriamente dita, sendo responsável pela conversão dos formatos de dados manipulados a nível de linguagem para um formato a nível operacional, permitindo envio de invocações com parâmetros e retorno de resultados.

Os objetos procuradores são manipulados como objetos locais e garantem a transparência de ações do sistema distribuído. Toda invocação a um objeto em outra unidade de endereçamento, outro nodo processador ou simplesmente outro processo do mesmo nodo, é feita indiretamente através do objeto procurador. A figura 4.11 apresenta o processo de invocação de um método do objeto **A** pelo objeto **B**.

A figura 4.11 mostra o caminho da mensagem de invocação desejado e, abaixo, o caminho realmente percorrido através do objeto procurador. As mensagens com retorno de resultado percorrem o mesmos caminho das invocações, porém em sentido contrário.

4.2.5 Mensagens entre objetos

Através da malha de comunicação que conecta os nodos processadores trafegam dois tipos de mensagens: mensagens com invocação de métodos e mensagens de resposta de objetos que tiveram métodos invocados. Estas mensagens permitem realizar o sincronismo entre objetos, de acordo com os tipos de

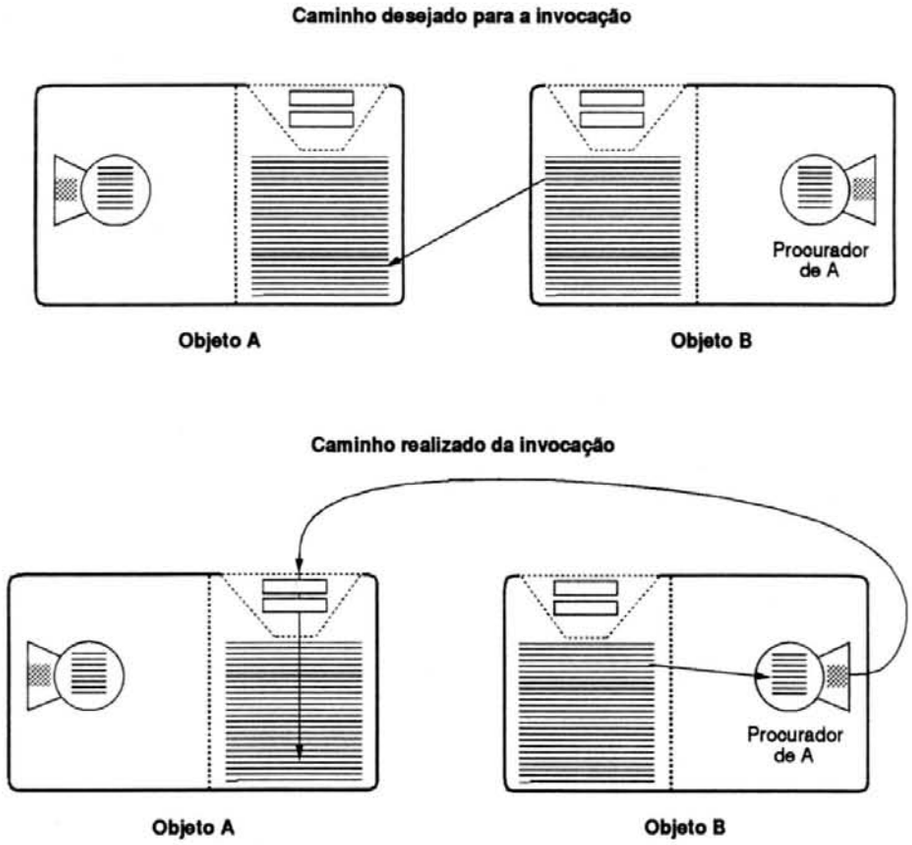

Figura 4.11: Processo de invocação de métodos através de objeto procurador

retorno de resultados permitidos pelos métodos. Abaixo encontram-se descritas as formas de sincronismo permitidas pelo modelo.

Invocações assíncronas . Este tipo de mensagem é enviada a métodos sem retorno de resultados, permitindo obter um grau máximo de concorrência entre objetos. Um objeto envia uma mensagem invocando algum método em outro objeto, prosseguindo imediatamente sua execução, não aguardando nenhum retorno de resultado.

Invocações assíncronas com confirmação . A confirmação do recebimento de uma invocação é desejável quando se faz necessário o conhecimento do momento em que um serviço requisitado é iniciado. Como no caso de invocações assíncronas, tanto o objeto origem da mensagem como o objeto recipiente executam suas tarefas de forma concorrente. A diferença está em que o objeto origem permanece bloqueado até

receber uma mensagem que confirme o início do tratamento da mensagem enviada.

Invocações síncronas . Diferente das demais, um objeto ao enviar uma invocação a um método com retorno de resultados permanece bloqueado, até que o objeto recipiente termine o tratamento da mensagem enviada e retorne uma mensagem de resposta. Com este tipo de invocação não existe concorrência durante a execução dos objetos envolvidos, mas há a possibilidade de retorno de resultados.

4.2.5.1 Pacotes de mensagens

Para trafegar através da malha de comunicação, as mensagens de invocação e de resposta são inseridas em pacotes de comunicação. Estes pacotes de comunicação são mostrados na figura 4.12.

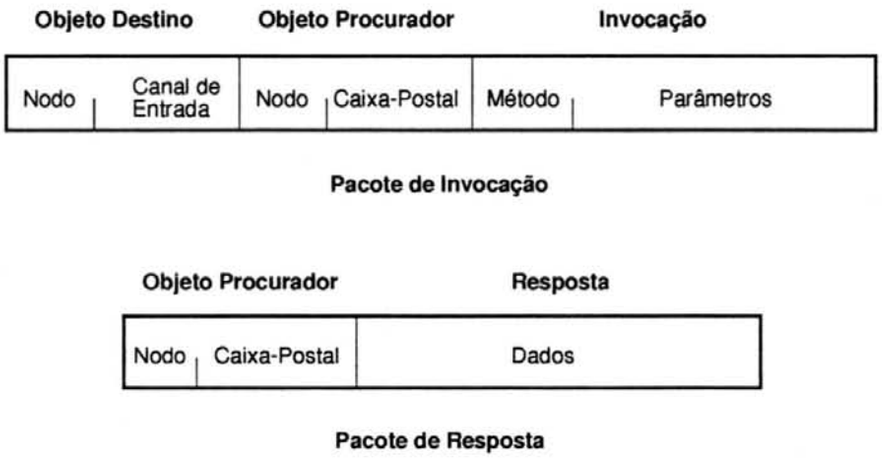

Figura 4.12: Pacotes de comunicação de mensagens

Os pacotes de invocação são montados e enviados pelo objeto através de objetos procuradores. Este processo é ativado quando o objeto ao qual um objeto procurador está associado envia uma invocação indicando o método e os parâmetros necessários ao objeto distribuído que representa. Os campos do pacote de invocação são os seguintes:

- **Objeto destino**: este campo é necessário para que os elementos de comunicação possam enviar os pacotes para os objetos destinatários corretamente. O subcampo **nodo** identifica o nodo processador que suporta a execução do objeto desejado e o subcampo **canal de entrada** identifica a fila de mensagens associada a interface de entrada do objeto destinatário.

- **Objeto procurador**: este campo, manipulado pelo servidor de mensagens do objeto recipiente, permite o endereçamento de pacotes com mensagens de respostas às mensagens recebidas. O subcampo **nodo** indica o processador responsável pela execução do objeto origem do pacote e o subcampo **caixa-postal** a fila de mensagens onde este objeto aguarda a resposta.

- **Invocação**: subdividido em dois subcampos, **método** e **parâmetros**, este campo é utilizado pelo elemento de delegação do objeto recipiente, permitindo identificar o método desejado e realizar a passagem de parâmetros.

A responsabilidade da montagem e do envio do pacote de resposta cabe ao servidor de mensagens do objeto invocado. Os destinatários dos pacotes de resposta são os objetos procuradores que enviaram invocações de métodos. Uma vez recebido um pacote de resposta, o objeto procurador retorna os dados recebidos ao objeto associado, após a devida conversão de formato. Os campos deste pacote são:

- **Objeto procurador**: contém informações de endereçamento do pacote. O subcampo **nodo** identifica o nodo processador responsável pela execução do objeto procurador e o subcampo **caixa-postal**, a fila de mensagens associada com a caixa-postal do objeto procurador.

- **Resposta**: este campo consiste nos dados retornados pelo método ativado no objeto remoto.

4.2.6 Diretório

O *Diretório* consiste no elemento do modelo cuja capacidade de serviços pode ser adaptada de forma a atingir diversos tipos de aplicações. Sua tarefa básica é o controle dos objetos e dos nodos de processamento. Para tanto, o Diretório manipula duas tabelas, a tabela de identificação de objeto e a tabela de controle de nodos, cujas entradas estão esquematizadas na figura 4.13.

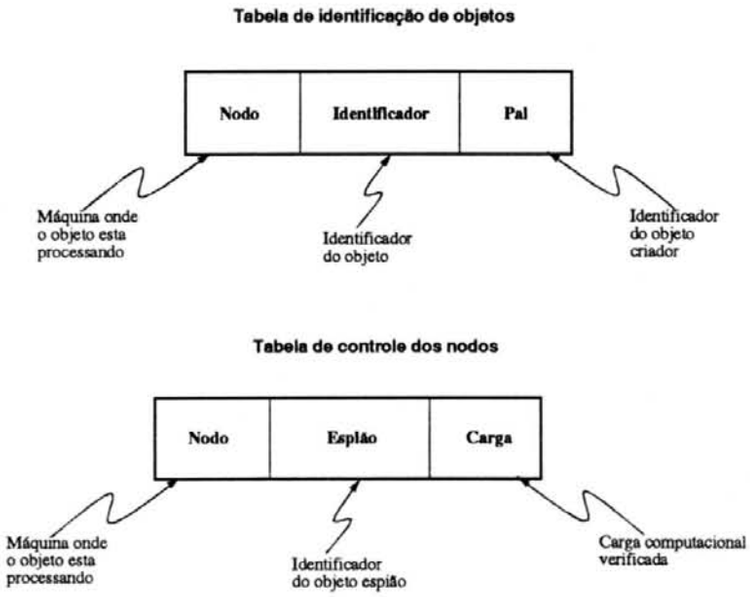

Figura 4.13: Tabela de identificação de objetos

A implementação do Diretório é através de um objeto distribuído, portanto, o acesso aos seus serviços é feito com invocações a seus métodos através de objetos procuradores. Novas funções para o Diretório podem ser facilmente inseridas através da adição de novos métodos.

4.2.6.1 Funções básicas do Diretório

Os serviços indispensáveis ao Diretório consistem nas tarefas de criação e remoção de objetos. Estas operações são possíveis devido ao Diretório ter acesso a uma unidade de armazenamento contendo o código executável dos objetos. A operação de criação conta também com um serviço de controle de carga computacional dos nodos disponíveis para processamento, possibilitando instanciar o novo objeto no nodo processador com menor carga computacional.

Controle dos nodos

O Diretório mantém atualizada uma tabela de controle de carga com vistas a distribuir melhor a carga computacional da aplicação, tirando proveito dos nodos com baixa ocupação. O mecanismo utilizado consiste em instanciar em cada nodo do ambiente um *objeto espião*. Este objeto espião mantém o controle sobre a taxa de utilização da UCP do nodo em que está executando.

Os objetos espiões possuem a mesma configuração dos objetos distribuídos, sendo que seus objetos procuradores são mantidos pelo Diretório. O único método deste objeto retorna a taxa de utilização da UCP no momento da solicitação. Uma consulta a todos os objetos espiões feita pelo Diretório mantém a tabela de controle de carga computacional atualizada.

Criação de objetos

A criação de objetos no modelo envolve duas etapas, a primeira executada localmente e a segunda de forma remota. A nível local ocorre a invocação do método construtor de um objeto **B** por um objeto **A**, sendo instanciado um objeto procurador de **B**. Com o objeto procurador se dá o início da etapa remota

da operação. Esta segunda etapa consiste na criação de um novo processo para suportar a execução do objeto **B** através do Diretório.

Os passos envolvidos na etapa remota são:

1. envio de uma mensagem do objeto procurador ao Diretório solicitando a criação do objeto remoto;
2. criação, pelo Diretório, do novo objeto, buscando em uma unidade de armazenamento externa o código que o implementa;
3. adição de uma nova linha na tabela de identificação de objetos caracterizando o objeto criado;
4. retorno ao objeto procurador da identificação do objeto criado. Esta identificação é composta de uma tupla contendo o canal de entrada e o nodo processador do objeto; e,
5. finalmente, o objeto que requisitou a criação é liberado para prosseguir sua execução, podendo utilizar os serviços do objeto remoto através do objeto procurador de forma transparente.

O passo 2, criação do novo objeto, consiste na criação pelo Diretório de um novo processo, executando o código do objeto criado, em um dos nodos processadores do ambiente. A escolha do nodo que irá suportar a execução do objeto pode ser *automática* ou *manual*. Sendo manual, deve ser definido pelo programador o nodo que deverá suportar a execução do objeto criado. Este recurso é utilizado quando a opção por algum nodo processador levar em consideração alguma especialização funcional implementada em hardware, por exemplo um objeto requerer saída gráfica em um vídeo colorido não disponível em todos os nodos. A indicação do nodo desejado é feita no momento da invocação da operação de criação do objeto.

Caso não seja especificado nenhum nodo processador, o Diretório pesquisa na tabela de controle de carga o nodo com menor carga ocupacional. Este nodo é escolhido para suportar a execução do objeto a ser criado. Como resultado da alocação automática de processadores é evitado a sobrecarga de nodos processadores e a conseqüente degradação de desempenho.

Remoção de objetos

Um objeto é removido do ambiente de execução em duas situações: ao receber uma mensagem indicando seu término ou quando seu objeto criador, objeto-pai, for removido. A operação de remoção de objetos é realizada através do Diretório, sendo possível manter a integridade do controle de objetos.

A remoção de objetos por meio de mensagem *explícita*, pelo envio de uma mensagem a um método destrutor, somente pode ser realizada de um objeto-pai para um objeto-filho. Este controle é feito pelo Diretório, visando impedir ações corruptas durante a execução do programa, permitindo somente ao objeto criador direitos sobre o momento de remoção do objeto-filho.

A remoção de toda árvore que compõe os objetos-filhos de forma *implícita*, quando ocorre a remoção de um objeto, é efetuada para que não ocorra a existência de "objetos-zumbis", uma vez que estes não poderão ser removidos de forma explícita. Esta operação também possibilita a realização de uma espécie de *garbage collection*, evitando a permanência de objetos não utilizados no ambiente de execução.

4.2.6.2 Outras funções para o Diretório

O controle dos objetos e dos nodos de processamento é o recurso mínimo para garantir o processamento distribuído em uma linguagem utilizando

o modelo proposto. Tirando proveito da característica de controle centralizado provido pelo Diretório, é possível implementar operações que viabilizem uma maior abrangência de tipos de aplicações ou que aumentem o desempenho e a confiabilidade da linguagem que implementa o modelo.

A seguir estão descritos três serviços, migração de objetos, tolerância a falhas e servidor de objetos, que podem ser implementados pelo Diretório, com ou sem participação de outros componentes do modelo. A opção em implementar estes serviços depende do uso ao qual será dado à linguagem que incorporar o modelo.

Migração de objetos

A migração de objetos favorece a execução de aplicações cujos objetos possuam altas taxas de processamento. O Diretório, por ter capacidade de verificar a utilização da UCP de cada nodo, pode detectar nodos com carga excessiva, retirando destes objetos para ser alocado em um nodo com capacidade de processamento ocioso.

O problema de endereçamento de objetos "móveis" é solucionado através das mensagens síncronas e assíncronas com confirmação. Caso não haja recebimento de resposta a uma mensagem enviada no decorrer de um determinado tempo, o objeto procurador solicita ao Diretório a nova localização do objeto que representa. A partir deste momento, o objeto procurador passa a referenciar o objeto com seu novo identificador.

Mensagens assíncronas não poderão ser enviadas a objetos que possuam capacidade de migrar, uma vez que para este tipo de invocações não há qualquer controle sobre seu recebimento, podendo portanto serem perdidas.

Objetos estáticos, que não podem migrar de um nodo de processamento para outro, coexistem com objetos dinâmicos, devendo possuir atributos que os diferenciem. Este atributo é especificado pelo objeto-pai no momento da criação do objeto. Este atributo é registrado na tabela de identificação de objetos, possibilitando o controle pelo Diretório. A figura 4.14 apresenta a tabela de identificação de objetos estendida para suportar o controle de migração de objetos.

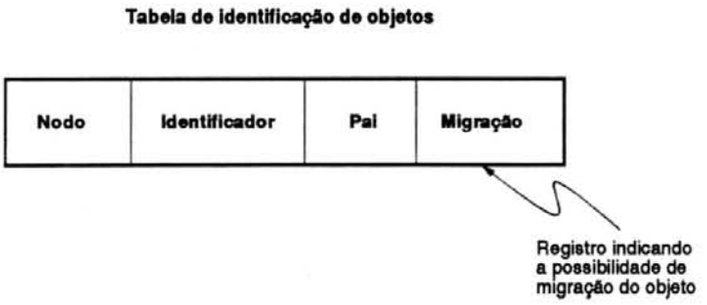

Figura 4.14: Tabela de identificação de objetos com controle de migração

Tolerância a falhas parciais

Para atender a aplicações que necessitem garantias da manutenção de estado íntegro de objetos e sua execução correta, o Diretório pode manter um esquema de arquivo de estados consistentes de objetos, provendo um esquema de tolerância a falhas. Estes arquivos são atualizados com os estados internos dos objetos em intervalos de tempos regulares controlados pelo Diretório. Para mensagens recebidas entre atualizações dos registros de estados consistentes de objetos é mantido um arquivo de *log*. Este *log* de mensagens é utilizado para recompor o objeto em caso de necessidade de recuperação.

A detecção da ocorrência de algum problema de execução em um objeto ocorre pelo não recebimento de resposta a uma mensagem síncrona ou assíncrona com confirmação. Neste caso, o objeto procurador que realizou a

chamada envia uma mensagem para o Diretório solicitando a recuperação do objeto.

Tendo o problema ocorrido por falha de um nodo processador, a detecção pode ocorrer no momento da solicitação das informações sobre a taxa de utilização para o objeto espião de um nodo. Neste caso é necessário que esteja implementado algum mecanismo de migração, possibilitando recuperar os objetos do nodo defeituoso em outro nodo processador.

A recuperação de um objeto consiste na criação de um novo processo, contendo o último estado consistente arquivado, sendo realizadas novamente as invocações armazenadas em *log*, devendo ser ignoradas os envios de mensagens de respostas. Deve-se notar que se a falha ocorrida em um objeto tenha sido resultado de um problema de implementação da aplicação, o processo de recuperação irá acarretar novamente na ocorrência da falha. Portanto, é possível apenas recuperar objetos cuja falha tenha ocorrido devido a fatores externos.

Servidor de nomes de objetos

Estendendo as funções de controle de objetos feita pelo Diretório é possível implementar um servidor de nomes de objetos. Com este recurso é possível implementar aplicações cooperativas ou um banco de serviços.

Para implementação deste recurso é necessária a introdução de dois novos campos na tabela de identificação de objetos, como mostra a figura 4.15: um nome "fantasia" para o serviço oferecido e um controle do número de acessos simultâneos permitidos a este serviço.

A criação de um objeto cujo serviço deva ficar disponível é acompanhado do nome fantasia que identifica o serviço prestado pelo objeto e o número de referências simultâneas que o objeto pode suportar. A solicitação de um serviço

Figura 4.15: Tabela de identificação de objetos para servidor de nomes

é indicada com uma mensagem enviada ao Diretório, que envia como resposta a identificação do objeto desejado. Uma vez atendida a solicitação, é decrementado o número de acessos simultâneos, realizando o controle de referências existentes ao objeto.

Não podendo ser atendida, a solicitação falha, sendo, neste caso, enviada uma mensagem ao objeto indicando a necessidade de uma nova tentativa em uma nova oportunidade. A solicitação não poderá ser atendida caso o serviço requisitado não esteja cadastrado ou, estando cadastrado, o número de acessos simultâneos esteja esgotado.

Um exemplo do uso deste recurso pode ser dado pelo serviço de impressão a ser oferecido por um objeto. Quando criado, e solicitado seu cadastro pelo Diretório com o nome fantasia **impressora**, possibilitando apenas um acesso por vez, este serviço fica disponível aos demais objetos que compõem a aplicação. Um objeto que desejar os serviços fornecidos pelo objeto **impressora** cria um objeto procurador para **impressora** o qual requisita ao Diretório a localização do serviço através do nome fantasia. Após o uso do serviço, o objeto destrói o objeto procurador de **impressora**, cuja última ação é liberar o recurso enviando uma mensagem para o Diretório.

A remoção dos objetos cadastrados como serviço ocorre de forma explícita, com mensagem enviada por seu objeto criador. No caso do objeto criador ser removido não ocorre a remoção implícita do objeto cadastrado como

prestador de serviço. A remoção de forma implícita ocorre somente no término da aplicação.

4.2.7 Ciclo de vida de um objeto

O ciclo de vida de um objeto distribuído no modelo apresentado é visto na figura 4.16, cuja duração máxima corresponde ao tempo de execução da aplicação. Além do ciclo de vida, a figura também representa a *classe distribuída* a partir da qual o objeto é instanciado.

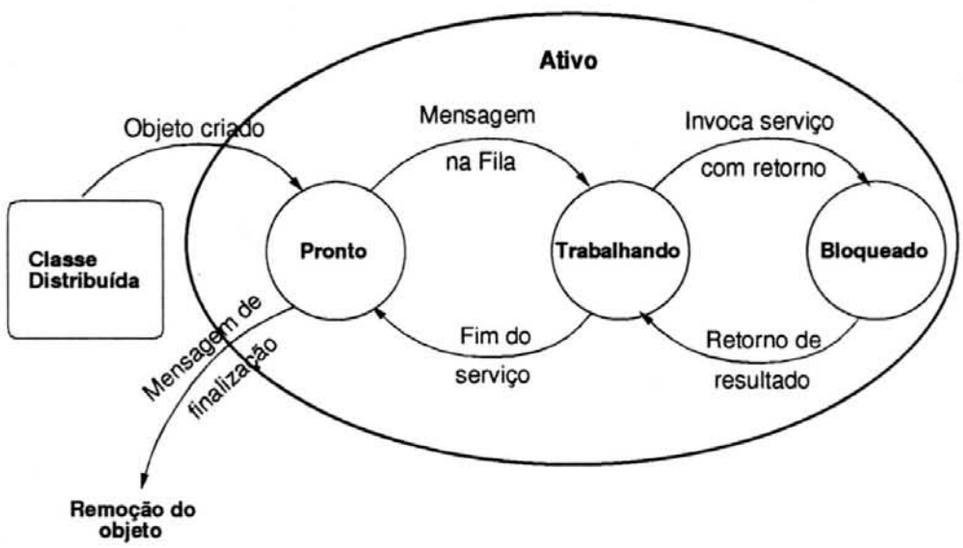

Figura 4.16: Ciclo de vida de um objeto distribuído

O estado *ativo* do objeto corresponde ao objeto instanciado a partir de uma classe distribuída. Este estado corresponde a execução propriamente dita do objeto, contendo seu próprio processo alocado a algum nodo processador. Neste estado, o objeto pode encontrar-se processando alguma mensagem recebida, aguardando alguma invocação ou o retorno de alguma mensagem síncrona ou assíncrona com confirmação enviada. A entrada de um objeto no estado ativo corresponde a sua criação, pelo Diretório, a partir de uma solicitação recebida. A

saída deste estado corresponde a remoção do objeto, sendo realizada pelo recebimento de uma mensagem também enviada pelo Diretório.

No estado *Pronto* o objeto encontra-se pronto a atender uma mensagem de invocação que esteja na fila de mensagens referente ao objeto. Caso a fila esteja vazia, o objeto permanece neste estado, caso contrário, a mensagem é retirada da fila, recebendo o tratamento adequado.

O estado *Trabalhando* corresponde ao processamento da invocação recebida. Enquanto o objeto estiver neste estado, é executado o código que implementa seus métodos e acessados os objetos locais. Sendo necessário a ativação de outro objeto distribuído para executar o serviço requisitado, o objeto envia mensagens através dos objetos procuradores. Caso estas mensagens sejam síncronas ou assíncronas com confirmação, o objeto é bloqueado até receber a resposta desejada. Uma vez terminado o atendimento à invocação recebida, o objeto está pronto a atender novas solicitações.

O objeto permanece no estado *Bloqueado* ao enviar invocações de métodos a objetos distribuídos. O objeto permanece neste estado enquanto o objeto procurador não receber retorno de resultados a uma mensagem enviada.

4.3 Nível de Linguagem

A escrita de um programa na linguagem distribuída não foge ao estilo de programação orientada a objetos. As restrições são encontradas na impossibilidade de uso de variáveis de classe e da passagem de parâmetros por referência, entre objetos distribuídos, devido a não existência de memória de uso compartilhado. Também devido a introdução do modelo distribuído se faz necessário a inclusão de novos elementos sintáticos, inexistentes em linguagens

seqüenciais, verificados em tempo de compilação indicando classes que definem objetos distribuídos e as formas de sincronização.

4.3.1 Restrições na manipulação de memória

Tratando-se de uma linguagem com endereçamento distribuído, não há possibilidade de objetos compartilharem uma mesma área de memória. Um objeto distribuído, quando instanciado, cria sua própria área de dados, inacessível aos demais objetos. Tal característica restringe o uso de alguns recursos disponíveis em linguagens orientadas a objetos seqüenciais.

Um esquema comum em linguagens orientadas a objetos é a utilização de variáveis de classe, consistindo em uma área de memória compartilhada por todos os objetos instanciados de uma mesma classe. Este esquema implica na possibilidade de endereçamento de memória comum pelos objetos, característica inexistente no ambiente distribuído proposto.

Outra restrição é a impossibilidade de passagem de parâmetros por referência à área de memória ocupada por um objeto, sendo possível apenas a passagem de parâmetros por valor. Mesmo restritiva para muitas aplicações, esta característica é bastante adequada ao paradigma de orientação a objetos, onde a um objeto não é permitido o acesso direto a dados externos.

Mesmo em linguagens onde é permitido a passagem de parâmetros por referência, recomenda-se sua não utilização. Este é o caso de C++, que permite a passagem de parâmetros através de seu endereço de memória, cujo conteúdo pode ser alterado, mas regras de programação aconselham a evitar seu uso [NYQ92], pois somente o objeto pode manipular seus próprios dados.

A mesma argumentação pode ser utilizada nos casos de retorno de resultados de métodos através de endereços de dados locais.

4.3.1.1 Endereços de localização

No modelo proposto há dois tipos distintos de objetos, locais e distribuídos. Objetos locais pertencem a um *cluster* de um determinado objeto distribuído. Entre objetos locais, que compartilham o uso de um módulo de memória, a identificação de localização é provida pelo endereço da porção de memória que ocupam.

A identificação de objetos distribuídos é feita através de objetos procuradores. Estes objetos procuradores manipulam uma *tupla* contendo o nome da máquina que está executando o objeto que representa e seu canal de entrada. As mensagens são endereçadas pelos objetos procuradores através do uso desta tupla, que identifica a localização dos objetos que representam.

Entre objetos é possível transferir identificadores de objetos, permitindo a comunicação no sistema. Porém, enquanto entre objetos locais é possível transferir identificadores de objetos locais e distribuídos, entre objetos distribuídos somente é possível a transferência de identificadores de objetos distribuídos. Neste último caso, a transferência de indentificador é feita por passagem de parâmetros, sendo o elemento de Delegação do objeto responsável por criar um objeto procurador para referenciar o objeto remoto.

4.3.2 Diretivas de compilação

Em linguagens orientadas a objetos seqüenciais todos os objetos consistem em objetos locais, que podem ter uso compartilhado pelos demais através

da memória, existindo um único fluxo de execução em todo o programa. Neste tipo de linguagem, a granulosidade de execução da aplicação é igual a granulosidade do programa. No modelo distribuído, além de objetos locais, existem objetos distribuídos em execução. O programador deve diferenciar as classes que definem objetos locais de objetos distribuídos, definindo a granulosidade de execução do programa.

Assumindo que todas as definições de classes são para objetos locais, uma classe para objetos distribuídos deve possuir um atributo extra. Por exemplo, em C++, a palavra reservada *class* identifica a definição de uma classe; sendo o modelo inserido nesta linguagem, a palavra reservada *class* é utilizada para definição de objetos locais, exceto quando precedida do atributo *distributed*, compondo *distributed class*, o qual indica a definição de uma classe para objetos distribuídos [CAV93a].

A criação de instâncias de classes de objetos locais e de classes de objetos distribuídos, bem como o envio de invocações, possuem as mesmas expressões sintáticas. Assim sendo, o acesso a objetos distribuídos é realizado de forma totalmente transparente, independente da localização do objeto remoto. A única exceção é no momento da criação de um objeto distribuído, em que pode ser enviado um parâmetro extra para o método construtor da classe indicando o nodo em que deseja-se instanciar o objeto.

Os diferentes tipos de retorno de métodos também sugerem a necessidade de uma caracterização a nível sintático. Desta forma é possível que o programador identifique a forma de interação entre os métodos, garantindo a expressão de pontos de sincronismos e de processamento concorrente.

Novamente tomando o exemplo de C++, métodos sem retorno de resultados são identificados pela palavra reservada *void* e métodos com algum resultado são identificados pelo tipo de dado retornado. Sendo o modelo proposto introduzido em C++, é necessário apenas criar uma identificação sintática para

métodos com confirmação de recebimento de mensagens. Isto é feito adicionando uma nova palavra reservada: *confirmation*.

As demais construções a nível sintático, como implementação de métodos e operações matemáticas, são estruturadas na linguagem distribuída como em uma linguagem seqüencial.

A figura 4.17 apresenta um exemplo de definição de classe distribuída em C++ supondo esta suportar o modelo proposto. A classe *Valvula* define objetos distribuídos que possuem associados os dados locais *vazao* e *abertura*, como representação do estado interno (objetos locais). Os métodos *Valvula* e *~Valvula* correspondem, respectivamente a criação (construtor da classe) e a remoção (destrutor da classe) de objetos desta classe, sendo *Valvula* executada como uma chamada síncrona, por ser o método construtor, e *~Valvula* como uma chamada assíncrona.

O método *Abrir* tem a função de abrir a válvula representada pelo objeto com um ângulo recebido como parâmetro. Por ser definido como *void*, o método *Abrir* recebe mensagens assíncronas. O método *Fechar* envia uma mensagem de confirmação garantindo o início da execução da tarefa – fechar a válvula a que está associado. Já o método *Abertura* envia como resposta um valor inteiro, representando como dado de retorno o ângulo de abertura da válvula, caracterizando seu uso associado a uma mensagem síncrona.

4.3.3 Herança

O mecanismo de herança encontra-se disponível no modelo a nível de linguagem, podendo ser utilizado tanto na descrição de objetos locais como de objetos distribuídos. O modelo não restringe o uso de herança simples ou

```
distributed class Valvula
                {
                        int     vazao,
                                abertura;

                    public:
                                Valvula   ( void );
                                ~Valvula  ( void );

                        void            Abrir       ( int );
                        confirmation    Fechar      ( int );
                        int             Abertura    ( void );
                }
```

Figura 4.17: Exemplo de uma classe distribuída em C++ suportando o modelo

múltipla. A implementação da herança, contudo, difere quando utilizada em classes locais e classes distribuídas.

A herança utilizada em classes de objetos locais é implementada através de *compartilhamento* de código. Evita-se assim o uso excessivo de memória de um nodo replicando o código de vários objetos locais de uma mesma classe em um mesmo cluster. Já para instâncias de classes de objetos distribuídos, o código deve ser replicado, existindo uma cópia completa da hierarquia de especialização especificada.

Acompanha a replicação do código das classes de objetos distribuídos os códigos das classes de objetos locais instanciados direta ou indiretamente a partir desta classe. Esta replicação é necessária devido a todos os dados necessários a um objeto estarem alocados na mesma área de endereçamento de memória.

Apesar do consumo de memória que representa a replicação de código para cada objeto criado, o mecanismo de cópia é o utilizado nas linguagens orientadas a objetos concorrentes. Outras abordagens baseiam-se em delegação, mecanismo do modelo de Atores, ou simplesmente não implementam este recurso.

A opção por utilizar herança a nível de linguagem foi baseada no desejo de oferecer bons recursos de desenvolvimento de aplicações, mesmo havendo

um custo, conhecido a priori, na utilização de memória quando da execução da aplicação.

4.4 Geração do Ambiente de Execução

O ambiente de execução de uma aplicação distribuída, utilizando uma linguagem orientada a objetos com o modelo proposto, é obtido através dos seguintes passos:

1. modelagem do problema segundo o paradigma de orientação a objetos;
2. identificação de classes cuja execução dos objetos possa ser concorrente;
3. identificação de formas de interação entre os objetos;
4. programação da aplicação segundo os critérios do modelo a nível de linguagem;
5. introdução do modelo operacional; e,
6. geração do código executável.

A modelagem do problema segundo o paradigma de orientação a objetos permite o acesso pelo programador a uma ferramenta eficiente para construção de sistemas, podendo ser utilizados recursos como reutilização de códigos e realizar manutenções de forma simplificada. Com a introdução do modelo distribuído, a modelagem da aplicação desenvolvida deve-se preocupar em também determinar as classes distribuídas.

Para as classes distribuídas também devem ser definidos os tipos de interação entre os métodos, garantindo através dos três tipos de mensagens, um maior ou menor grau de concorrência.

O passo 4 refere-se a programação propriamente dita da aplicação, após as etapas de modelagem realizada nos passos 1, 2 e 3. A programação do sistema é feita através de uma linguagem distribuída, desenvolvida segundo o modelo a nível de linguagem. Esta linguagem implementa internamente o modelo operacional, podendo ter sido desenvolvida especialmente para suportar o modelo ou uma extensão de uma linguagem seqüencial já existente.

A introdução do modelo operacional e a geração do código executável, passos 5 e 6 respectivamente, são realizados através de um processo de compilação e pelo ambiente de execução.

4.4.1 Processo de compilação

O objetivo do processo de compilação é adicionar ao programa, escrito segundo o modelo a nível de linguagem, os recursos do nível operacional. Este processo é visto, de forma simplificada, na figura 4.1, sendo detalhado pela figura 4.18.

Ao compilador é submetido um programa contendo as definições de classes de objetos locais e distribuídos. Como resultado do processo de compilação são obtidos *clusters* de execução.

Cada cluster consiste em uma unidade autônoma de processamento, sendo composta pelas classes que implementam um objeto distribuído, objetos locais utilizados e dos objetos procuradores. Também faz parte do cluster os códigos que implementam as tarefas da interface de acesso do objeto.

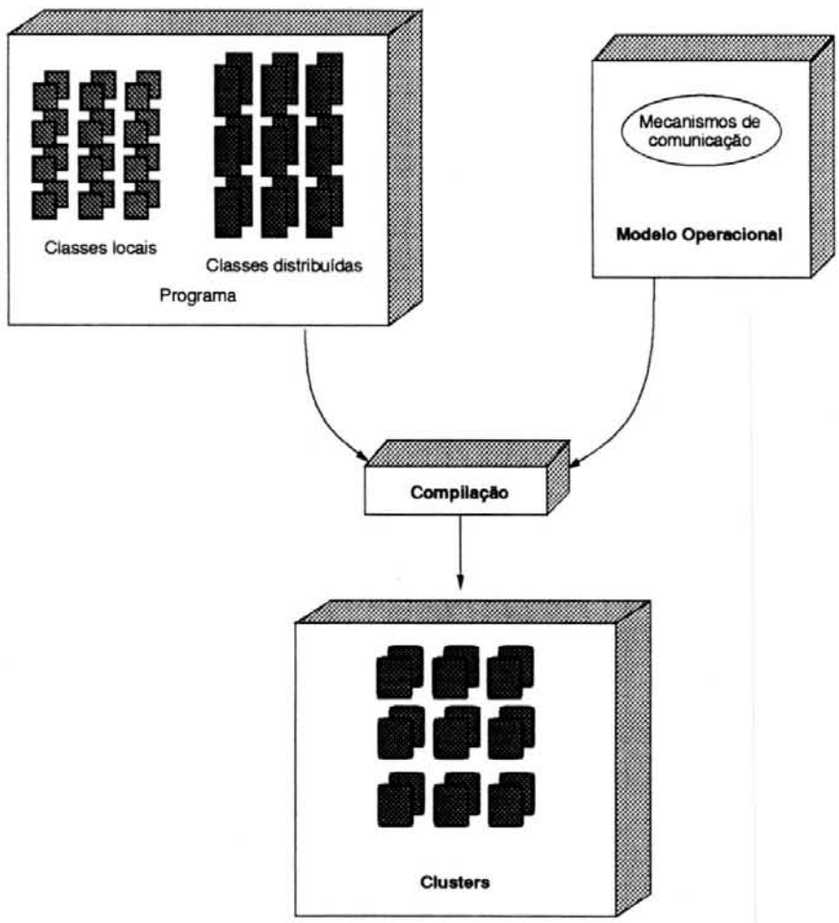

Figura 4.18: Processo de compilação dos clusters de execução

O código que implementa as classes dos objetos locais e distribuídos são obtidos diretamente do programa submetido ao compilador. O código das classes dos objetos procuradores, ou seja a definição e implementação das classes procuradoras, é obtido através de uma análise das classes distribuídas. Esta análise compõe para a classe procuradora uma mesma interface da classe distribuída, mas com a implementação dos métodos realizando um processo de captura e redirecionamento das invocações. As classes procuradoras são montadas de acordo com a definição das classes distribuídas, sendo gerada de forma automática uma classe procuradora para cada classe de objeto distribuído.

Dentro de um cluster, as instâncias de classes procuradoras representam um objeto distribuído instanciado a partir da classe original de objeto

distribuído. Desta forma é possível que acessos a objetos remotos sejam efetuado como se estes estivessem disponíveis localmente.

Os códigos que implementam os serviços da interface de acesso são introduzidos pelo compilador de forma automática. As regras obedecidas seguem as necessidades dos mecanismos de comunicação do modelo operacional. Este código implementa os recursos de tratamento de mensagens e de ativação de métodos conforme descritos anteriormente.

4.4.2 Ambiente de execução

O ambiente de execução provê suporte ao processamento da aplicação. Este ambiente é provido por um carregador de programa que recebe como entrada um arquivo *descritor* e os clusters de execução. O arquivo descritor contém a discriminação das classes distribuídas que compõe a aplicação, os nodos processadores a serem utilizados e a classe distribuída por onde deve iniciar a execução pela criação do primeiro objeto.

O arquivo descritor é utilizado para montagem do Diretório e alocação dos objetos espiões. O processo envolvido pelo ambiente de execução é visualizado na figura 4.19.

4.4.3 Concorrência e sincronismo

A unidade de processamento do modelo é o objeto distribuído. A granulosidade das operações realizadas depende do tamanho do grão de processamento em que foram modeladas as classes distribuídas. Em outras palavras, o próprio programador define a granulosidade de processamento quando definir o

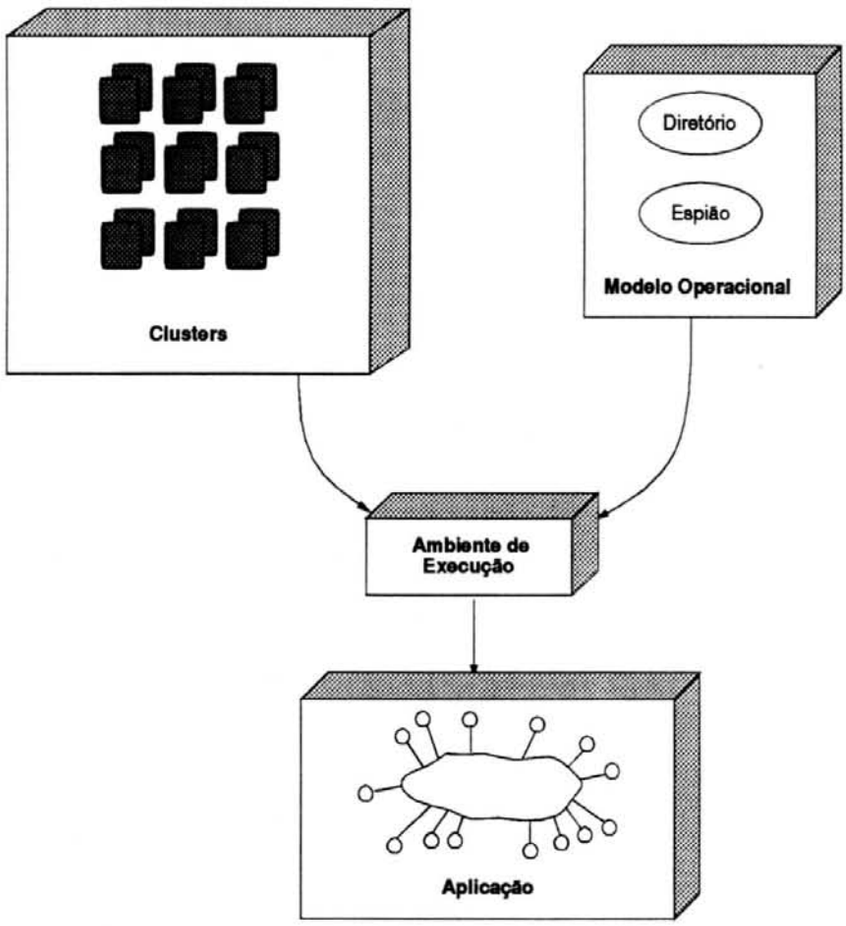

Figura 4.19: Suporte à execução do modelo de objetos distribuídos

modelo de uma classe. O sincronismo entre os objetos se dá através da troca de mensagens que representam o envio de invocações e retornos de resultados. O tratamento das mensagens pelos objetos distribuídos é realizado tratando uma a uma as invocações recebidas, respeitando a ordem de chegada.

Enquanto que entre os objetos distribuídos as invocações podem ser assíncronas, com ou sem confirmação, podendo ativar diversos fluxos de execução concorrentes, entre objetos de um mesmo cluster as invocações são sempre síncronas. Desta forma, dentro de um cluster o fluxo de execução é único.

A definição da sincronização entre objetos, a exemplo da definição do grão de processamento, é embutida na descrição da classe distribuída, sob forma de tipos de retorno de métodos.

4.5 Suporte ao Modelo

O modelo apresentado neste capítulo é suficiente no que se refere a execução distribuída de objetos a partir de uma linguagem orientada a objetos distribuída. Porém, outros recursos são necessários para efetivar sua implementação.

1. **Linguagem de programação**. O modelo prevê sua implementação suportada por uma linguagem de programação orientada a objetos. Esta linguagem pode ser uma linguagem nova ou uma das linguagens orientadas a objetos seqüenciais disponíveis adaptada para suportar o nível operacional do modelo. A primeira abordagem favorece a obtenção de melhores desempenhos para aplicações, dispondo de uma linguagem desenvolvida especialmente para suporte do modelo. Estender uma linguagem já difundida e conhecida pelos programadores simplifica sua utilização, favorece a migração de software, além de ser possível utilizar o ferramental já disponível para esta linguagem.

2. **Escalonamento de objetos**. Existindo a necessidade de dois ou mais objetos compartilharem um mesmo nodo, deve ser provido um serviço de escalonamento de objetos para uso da UCP. Sendo que o modelo prevê prioridades iguais para todos os objetos, o serviço de escalonamento é bastante simples, podendo ser implementado distribuindo fatias iguais de processamento (política de *round-robin*).

3. **Mecanismo de comunicação**. Os nodos de processamento devem oferecer mecanismos de trocas de mensagens entre objetos. O elemento de comunicação implementado nos nodo processadores deve possuir a capacidade de enviar e receber mensagens através da malha de comunicação escolhida para implementar o hardware de suporte. Também deve conter mecanismos de gerência de filas de mensagens

individuais para cada objetos que nele tenha sua base de execução. Filas únicas de mensagens devem ser providas também para os objetos procuradores.

4. **Acesso a uma base de clusters**. Os clusters executáveis dos objetos encontram-se armazenados em uma memória secundária gerenciada por um dos nodos processadores. Como instâncias dos objetos podem ser criadas nos diferentes nodos, esta memória secundária deve ser acessível a todo conjunto de processadores. O serviço de gerenciamento da memória secundária deve possuir mecanismos para atender solicitações de criação remota de processos executando o código do cluster correspondente ao objeto instanciado.

5. **Controle de utilização das UCPs**. É necessário um algoritmo que implemente o controle das taxas de utilização das UCPs dos nodos processadores para ser implementado pelos objetos espiões.

4.6 Sumário

Esta seção apresenta, em forma de tabelas, as características do modelo apresentado neste trabalho. A tabela 4.2 aborda os esquemas de concorrência e sincronização utilizados para controle da execução da aplicação. A tabela 4.3 apresenta as características da modelagem dos objetos disponíveis no modelo. Outras características do modelo encontra-se sumarizadas na tabela 4.4.

Tabela 4.2: Concorrência e sincronização

Característica	Implementação
comunicação	invocações a métodos;
sinconização	métodos síncronos métodos assíncronos com confirmação; métodos assíncronos;
concorrência	inter-objetos;
escalonamento	prioridades iguais para objetos e mensagens;
processos	um processo por objeto distribuído;
criação de processos	na criação de objeto distribuído;

Tabela 4.3: Modelagem dos objetos

Característica	Implementação
Modelos de objetos a nível de linguagem	distribuídos; locais;
Modelos de objetos a nível operacional	distribuídos; locais; procuradores;
criação	explícita na instanciação;
remoção	explícita ou implícita; na remoção do pai;
ativação	invocação de métodos;
recebimento de respostas	retorno de métodos síncronos;
herança	cópia (objetos distribuídos); compartilhamento (objetos locais);

Tabela 4.4: Outras características do modelo

Característica	Implementação
tolerância a falhas	mensagens assíncronas com confirmação; extensão do modelo;
migração	extensão do modelo;
servidor de objetos	extensão do modelo;
passagem de parâmetros	por valor;
distribuição da carga	na criação do objeto;

5 IMPLEMENTAÇÃO DE UM PROTÓTIPO

Para analisar a aplicabilidade do modelo, e o desempenho que pode ser obtido, foi implementado uma aplicação sobre um protótipo de uma linguagem orientada a objetos distribuída. Esta linguagem é denominada **DPC++** – Processamento Distribuído em C++, voltada ao processamento distribuído em estações de trabalho Sun sobre o sistema operacional SunOS, Unix compatível [SUN90b].

Os recursos utilizados para implementação da linguagem encontram-se descritos na próxima seção deste capítulo. Na seção seguinte, seção 5.2, encontra-se descrito o processo de geração do ambiente de execução DPC++. Por fim é apresentada a aplicação implementada, bem como índices de desempenho obtidos.

5.1 Recursos Utilizados

Os recursos utilizados para implementação dos esquemas de distribuição de DPC++ foram retirados do ambiente de rede das estações de trabalho [SUN90a]. Neste ambiente várias ferramentas encontram-se disponíveis para suporte ao ambiente distribuído. Em [ROS91, MAR91, CAV92] encontram-se descritas algumas destas ferramentas. Partindo dos estudos dos trabalhos citados foram escolhidas para implementar o modelo ferramentas as que apresentassem as características desejadas para cada item abordado no modelo.

5.1.1 Suporte básico

O hardware de suporte ao protótipo consiste em estações de trabalhos Sun, provendo uma UCP e memória para objetos distribuídos, ligadas por uma rede local. Destas estações, uma dispõe de um disco rígido para o armazenamento dos clusters de execução. Um barramento único ao qual as estações encontram-se conectadas provê o meio para comunicação. Esta configuração para o hardware utilizado é vista na figura 5.1.

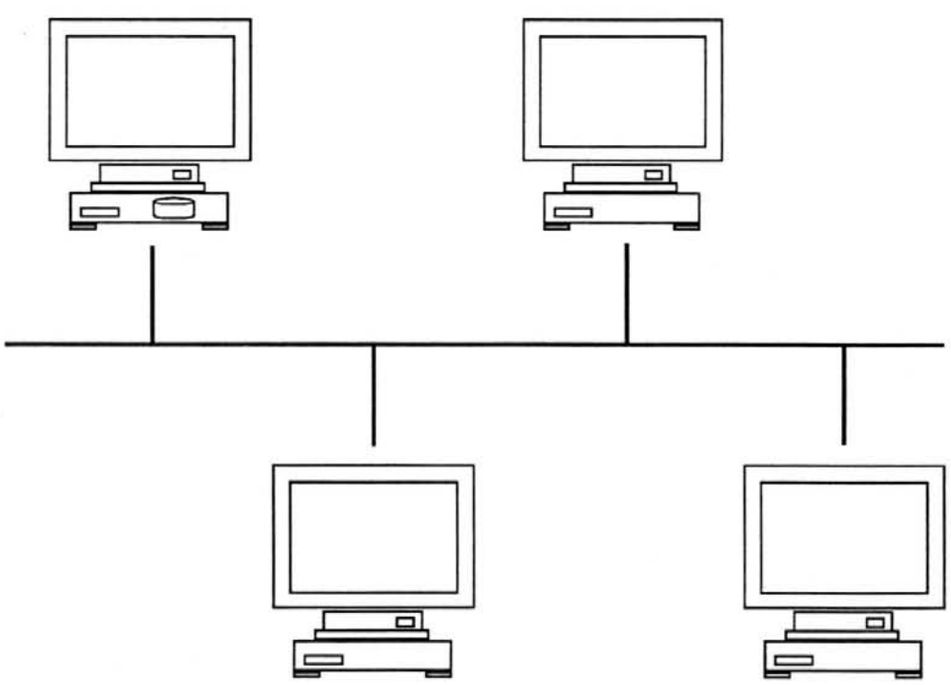

Figura 5.1: Rede de estações de trabalho

Os serviços de compartilhamento do nodo processador e o escalonamento de objetos para ocupar a UCP estão providos pelo SunOS, que executa sobre as estações. Também através do Unix é mantida a base de clusters executáveis e realizada a criação remota de processos.

5.1.2 Linguagem base

A linguagem protótipo DPC++ foi desenvolvida a partir de C++ [ELL90], como o próprio nome já sugere. DPC++ herda da linguagem base a estrutura de programação, a sintaxe, tipos básicos e, sobre tudo, as características da programação orientada a objetos. Um resumo das características de C++ pode ser encontrado no anexo A-1.

A programação em DPC++ é bastante próxima a programação C++. Na versão distribuída, porém, há a adição de duas novas palavras reservadas e restrições no uso de memória compartilhada. Com estas alterações, DPC++ tem o comportamento esperado pelo modelo a nível de linguagem.

As novas palavras reservadas são *distributed* e *confirmation*. Uma classe identificada com o atributo *distributed* corresponde a descrição de objetos distribuídos. A omissão deste atributo implica que a classe seja considerada de descrição de objetos locais.

Já a palavra reservada *confirmation* é utilizada para identificar métodos com confirmação de recebimento de mensagens. Métodos *void* e com algum tipo de retorno (*int*, por exemplo) são considerados métodos assíncronos e síncronos, respectivamente.

A figura 4.17, na seção 4.3.2, apresenta um exemplo de definição de classe distribuída em DPC++.

As restrições na manipulação de memória implicam na impossibilidade de qualquer forma de compartilhamento de memória, seja na passagem de parâmetros ou no compartilhamento de código ou dados. C++ permite três tipos de passagem de parâmetros: por referência, por endereço e por valor. Em DPC++, apenas esta última possibilidade é viável, uma vez que as outras duas pressupõem a existência de uma memória de uso comum.

No caso de passagem de estruturas do tipo *array* foi definido para DPC++ uma nova construção permitindo a passagem de *arrays* por valor. O uso desta estrutura é exemplificada abaixo com o cabeçalho de um método.

Arquivo :: GravaElementos (Elemento vet[10])

Neste exemplo, a classe distribuída *Arquivo* possui um método, denominado *GravaElementos* que deve gravar um vetor *vet* de 10 posições de *Elemento* recebido como parâmetro.

Uma chamada a este método possui a seguinte sintaxe:

```
Arquivo * arq = new Arquivo;
Elemento vetor[10];
.....
.....
arq->GravarElementos(vetor[0,9]);
.....
.....
```

Nesta chamada é invocado o método GravaElementos passando como parâmetro 10 elementos de *vetor*, da posição inicial 0 a final 9, do vetor de Elemento.

O compartilhamento de dados entre todas as instâncias de uma mesma classe em C++ é possível através do uso de *variáveis de classe*. O uso de variáveis de classe, definidas com o atributo *static*, pressupõem a existência de uma área de memória acessível por todos os objetos. Em DPC++ o recurso de variáveis de classe (*static*) não está disponível.

O compartilhamento de código em C++ é possível através de métodos *in-line*. A invocação a um método *in-line* em C++ implica, não na chamada de uma função, mas na reprodução do código que a implementa em tempo de compilação.

Como os métodos de um objeto acessam o estado interno do objeto, a utilização de métodos *in-line* pressupõem a necessidade de compartilhamento de memória. Portanto, em DPC++ as classes distribuídas não devem conter métodos *in-line*.

5.1.3 Mecanismo de comunicação

Para prover os mecanismos de comunicação entre os objetos foi utilizado o mecanismo de *sockets* [SUN90a]. Sockets, ou portas, são abstrações da arquitetura de comunicação, que suporta a criação e manutenção das conexões da rede em diversos domínios de endereçamento. O domínio AF_INET (AF corresponde a *Address Format*) utiliza endereços no formato Ethernet para comunicação entre processos.

Encontram-se disponíveis para comunicação entre processos dois tipos de sockets, o primeiro provendo uma ligação através de circuitos virtuais entre os pares da comunicação; este tipo de socket possibilita uma comunicação denominada orientada à conexão. O segundo tipo é baseado em datagramas, sem circuitos virtuais, são enviados pacotes pela rede contendo o endereço do destinatário. Ambas formas permitem a comunicação bidirecional entre processos, sendo a orientada à conexão mais confiável e a baseada em datagramas possibilitando melhor desempenho [VAI90].

Para utilização na implementação do protótipo foi escolhida a comunicação baseada em datagrama, devido as suas características suprirem as necessidades do modelo. Uma porta que permite a comunicação orientada a datagrama pode receber mensagens de qualquer outra porta com as mesmas características. Desta forma, um objeto pode ser acessado, através de mensagens, por qualquer objeto que possua seu endereço.

Se a comunicação entre objetos utilizasse circuitos virtuais isto não poderia ser conseguido, uma vez que as conexões são configuradas de forma estática, além de existir um número máximo de conexões por processo. Desta forma os objetos estariam limitados a um conjunto fixo de objetos com os quais poderiam comunicar.

A unidade de comunicação no mecanismo é o datagrama que consiste em um pacote individual de mensagem. Este pacote contem as informações de endereçamento do socket destinatário e do originador, além da informação transmitida. Assim, no recebimento de um datagrama é possível identificar o endereço de comunicação do processo originador da mensagem, viabilizando o envio de mensagens de respostas.

Ao contrário dos mecanismos orientados à conexão, o protocolo baseado em datagramas não é confiável, podendo ser perdidas ou duplicadas mensagens caso ocorra algum problema no meio físico de comunicação. Porém se o meio de comunicação for estável, com freqüência de falhas muito baixa, a probabilidade de erros no tráfego de datagramas é desconsiderável.

Os serviços que implementam os mecanismos de troca de mensagens estão disponíveis na classe *Comunicacao*, sendo utilizado pelos objetos procuradores e pela interface de acesso de objetos distribuídos. O anexo A-2 apresenta a classe *Comunicacao* comentando suas características.

5.2 O Ambiente de Execução

O ambiente de execução de uma aplicação escrita na linguagem DPC++ é obtida através do uso de um pré-compilador DPC++ e do compilador C++ [SUN89], como mostra a figura 5.2. O pré-compilador DPC++ gera, a partir do programa submetido, os clusters de execução e as funções de inicialização da

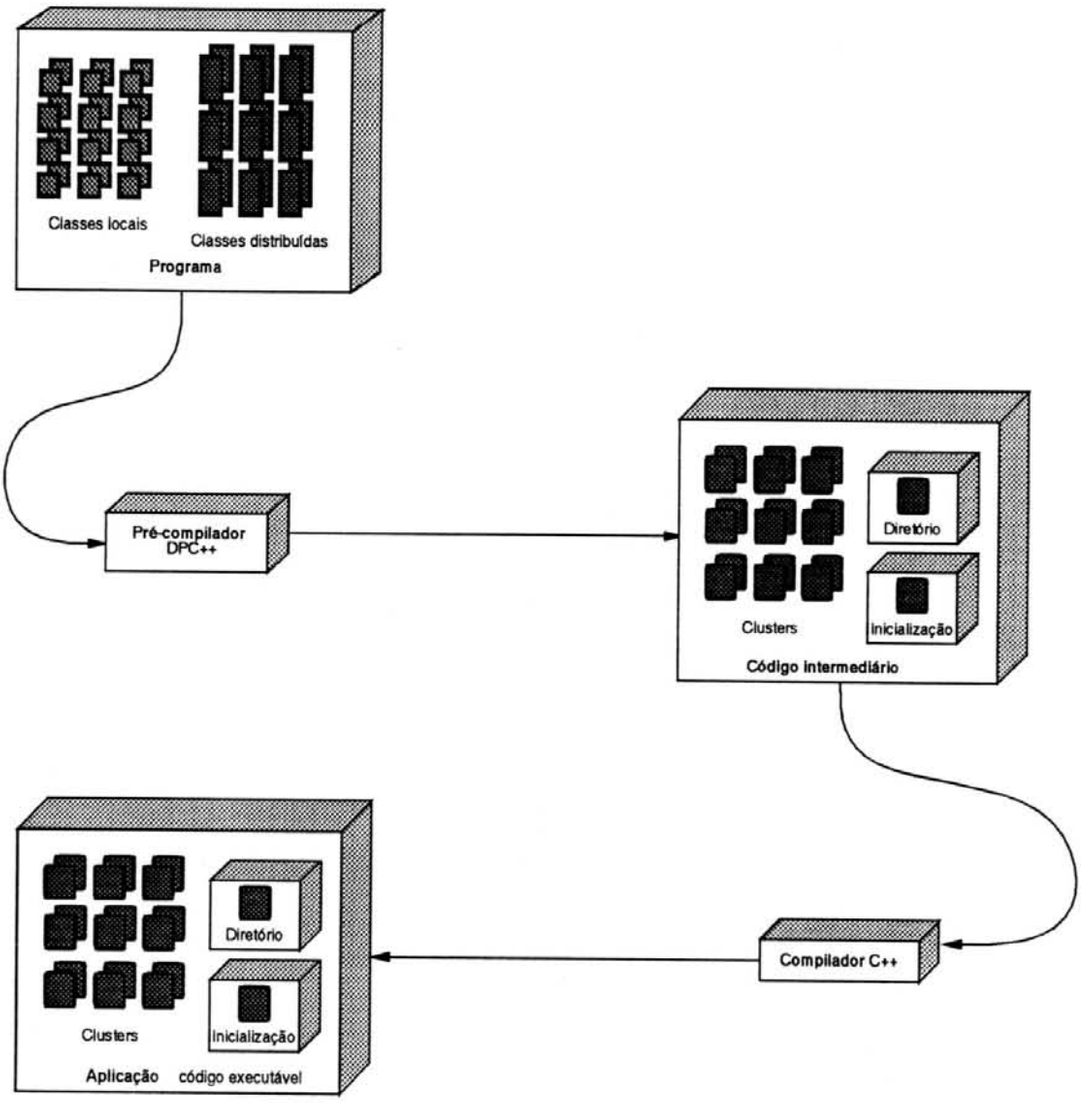

Figura 5.2: Geração do ambiente de execução

aplicação. As saídas do pré-compilador DPC++ são codificadas em C++, devendo ser submetidas ao compilador C++ para gerar o código executável.

A próxima seção descreve as características do pré-compilador DPC++. As seções restantes descrevem as estruturas geradas pelo pré-compilador para dar suporte a execução dos componentes do modelo operacional.

5.2.1 O pré-compilador DPC++

Cabe ao pré-compilador DPC++ montar o Diretório, os objetos procuradores e os clusters de execução de uma aplicação, partindo das classes distribuídas de um programa a ele submetido. São realizados também testes de verificação erros de sintaxe no programa. Caso o programa esteja correto sintáticamente, o resultado do processo de pré-compilação é um conjunto de arquivos contendo código em C++. Estes arquivos contém o trecho de código de inicialização da aplicação e dos clusters de execução, dos quais um é o cluster que implementa o objeto da classe Diretório.

As saídas do pré-compilador são submetidas individualmente ao compilador C++, gerando os respectivos códigos executáveis dos clusters.

Como a comunicação entre os componentes da aplicação se dá através de mensagens, o pré-compilador atribui *identificadores* para as classes distribuídas e seus métodos. Estes identificadores consistem em valores numéricos que identificam as diferentes classes e métodos, sendo utilizados para compor os campos dos pacotes de invocação de mensagens (figura 4.12).

Os *identificadores de classe* são enviados como parâmetros nas mensagens que solicitam a criação de objetos. Portanto, estes identificadores devem ser conhecidos tanto pelo objeto procurador que está requisitando a criação quanto pelo Diretório, que deve criar um objeto da classe desejada.

Os *identificadores de métodos* são utilizados pelos objetos procuradores e pelo serviço de delegação provido pela interface de acesso de um objeto distribuído. O objeto procurador preenche o sub-campo **método** do pacote de invocação com o identificador do método desejado. O serviço de delegação utiliza o código deste sub-campo nas invocações recebidas para identificar o método desejado.

5.2.2 Implementação dos clusters

Os clusters consistem em processos dedicados a suprir com recursos computacionais (capacidade de processamento e memória) as necessidades de um objeto distribuído. Os clusters são gerados a partir de classes distribuídas pelo processo de compilação com o compilador DPC++ ao final de duas fases. A primeira fase resulta nas classes procuradoras obtidas da definição das classes distribuídas. A segunda fase monta os clusters contendo:

1. o código de uma classe distribuída;
2. os códigos das classes de objetos locais utilizados pelos métodos da classe distribuída;
3. os códigos das classes de objetos procuradores de objetos distribuídos referenciados pelo cluster; e,
4. o código da interface de acesso.

Os dois primeiros itens são fornecidos pelo programa da aplicação, enquanto que os últimos são gerados pelo compilador a partir da definição das classes distribuídas. A estrutura das classes procuradoras e da interface de acesso são apresentadas a seguir.

5.2.2.1 Estrutura da classe procuradora

Uma classe procuradora é gerada a partir de uma classe distribuída. Possui métodos procuradores com o mesmo protocolo de acesso da classe que representa. Os métodos entretanto não executam as funções propriamente ditas, apenas montam pacotes de mensagens que são enviados aos objetos que irão executar a tarefa solicitada.

A estrutura da classe procuradora é apresentada na figura 5.3. Dois objetos de dados são internos aos objetos de classes procuradoras: o objeto *ServMsg* e o *DirProc*. O primeiro permite o envio de invocações ao objeto que representa, sendo o servidor de mensagens do objeto procurador. Já o objeto *DirProc* permite o envio de mensagens para o Diretório. Os demais componentes são os métodos procuradores, que recebem as invocações a serem enviadas a objetos distribuídos.

```
class clproc {

        Comunicacao    ServMsg;
        Diretorio      DirProc;

    public :

        <construtor>
        <destrutor>

        <demais métodos>
}
```

Figura 5.3: Estrutura da classe procuradora em DPC++

O método *construtor*, utilizando a característica de polimorfismo, pode ser especificado por vários métodos construtores para uma mesma classe. Cada um destes métodos vai receber, na classe procuradora, três versões. Uma das versões implica em uma simples criação do objeto distribuído. O método construtor procurador envia ao Diretório uma mensagem solicitando a criação do objeto distribuído. Na segunda versão, o construtor recebe um parâmetro extra, o qual identifica o nodo em que o objeto remoto deve ser instanciado. Nestes dois casos, tanto os objetos procuradores como o objetos distribuídos são criados no momento da invocação.

A terceira versão é utilizada quando o objeto distribuído já está criado e deseja-se criar apenas um objeto procurador. Neste caso, é passado por parâmetro ao construtor a tupla de identificação do objeto remoto. Esta tupla é utilizada para informar ao objeto *ServMsg* a localização do objeto que representa.

O método *destrutor* procurador é invocado quando deseja-se remover o objeto distribuído. A operação efetuada consiste no envio de uma mensagem assíncrona ao Diretório solicitando a remoção do objeto seguida da remoção do próprio objeto procurador. No caso de ter sido instanciado apenas o objeto procurador não é enviada a mensagem de remoção ao Diretório.

Os demais métodos recebem as invocações aos objetos distribuídos enviando-as de forma síncrona, assíncrona com confirmação ou assíncrona. Caso o método implique em uma chamada síncrona, o retorno do fluxo de execução do objeto procurador para o objeto originador da invocação somente irá ocorrer quando do recebimento dos dados de resposta, sendo possível seu repasse. No caso de métodos assíncronos com confirmação o retorno é efetuado no momento de recebimento da confirmação do recebimento da mensagem pelo objeto remoto.

Os métodos assíncronos liberam o fluxo de execução para o objeto invocador logo após o envio das mensagens de invocação.

A estrutura dos três tipos de métodos procuradores se encontra na figura 5.4. A tarefa de montar o pacote consiste em converter os parâmetros dos métodos do formato de linguagem para o formato de manipulação de informação a nível operacional. Adicionalmente é introduzido o sub-campo que identifica o método desejado.

5.2.3 Interface de acesso

A interface de acesso de um objeto implementa o serviço de ativação dos métodos através do elemento de delegação. O controle do recebimento de mensagens e envio de respostas é realizado pelo servidor de mensagens implementado pela classe *Comunicacao*, que é composta por métodos de manipulação

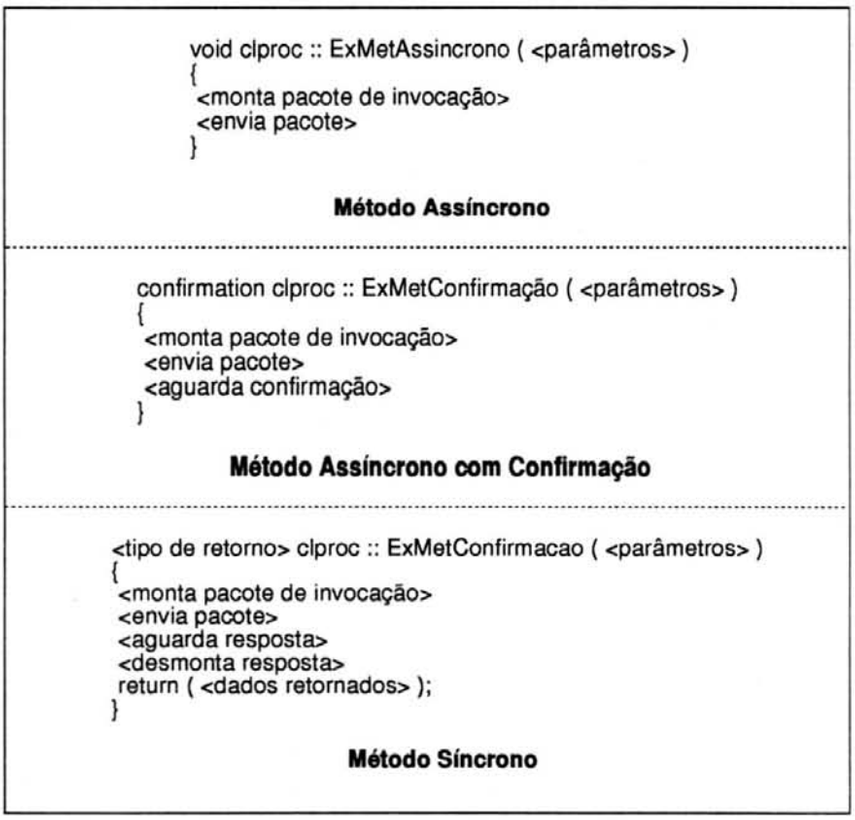

Figura 5.4: Estrutura dos métodos procuradores

de mensagens entre clusters. A interface de acesso de um objeto também é responsável por iniciar o processo de execução do cluster de que contém o objeto.

Como apresentada na figura 5.5, a estrutura da interface de acesso de um objeto é implementada na função principal do cluster, sendo portanto seu ponto de início da execução. O objeto ServMsg corresponde ao servidor de mensagens do objeto, enquanto que o serviço de delegação é implementado pela iteração infinita do algoritmo. Neste algoritmo, *CLDIST* corresponde a uma classe distribuída hipotética cujo objeto é executado pelo cluster.

A primeira tarefa da interface de acesso, quando em execução, é criar uma porta de comunicação para o objeto, através da criação de uma instância da classe Comunicação. Este endereço é enviado para o Diretório, permitindo que, posteriormente, o objeto receba mensagens de invocação através desta porta.

```
void main ( void )
{
 Comunicacao   ServMsg;
 CLDIST      * objdist;

 <cria porta de comunicação>
 <retorna endereço para o Diretório>

 while ( TRUE ) {
                <recebe pacote de invocação>
                <desmonta pacote de invocação>
                [<retorna confirmação>]
                <ativa método invocado>
                [<monta e envia pacote de resposta>]
 }
}
```

Figura 5.5: Algoritmo da interface de acesso

Abaixo estão descritos os passos que, executados infinitas vezes, compõem o serviço de delegação:

1. recebimento de um pacote de invocação de algum método;
2. desmontagem do pacote recebido, identificando o método desejado e os parâmetros;
3. caso o método seja do tipo *confirmation* retorna a confirmação do recebimento da mensagem;
4. ativa o método especificado; e,
5. caso tenha retorno de dados, é montado um pacote de resposta, sendo posteriormente enviado sobre o servidor de mensagens.

A iteração do algoritmo de delegação termina quando for recebida uma mensagem solicitando a execução do método destrutor do objeto. Após a execução do método destrutor, o processo que executa o cluster deixa de existir.

5.2.3.1 Implementação do Diretório

O Diretório possui a estrutura de um objeto distribuído comum do modelo, contando com os mesmos componentes: interface de acesso, métodos e área para objetos locais. Os objetos procuradores de objetos distribuídos da aplicação tem acesso aos serviços do Diretório é através de objetos procuradores pertencentes ao seu estado interno. A característica que diferencia o objeto Diretório dos demais é que sua implementação é conhecida a priori pelo ambiente de geração de compilação DPC++.

Sendo criado no momento da inicialização da aplicação, o objeto Diretório não possui um objeto-pai. Portanto para que os demais objeto tenham acesso a seus serviços, o endereço de localização do Diretório é especificado em tempo de compilação. Assim, os objetos procuradores do Diretório quando instanciados não requisitam a criação de um objeto remoto, mas passam a referenciar o objeto já criado.

Os métodos do Diretório são em número de três, um método construtor, um método destrutor e um método para atender solicitações de criação de objetos distribuídos. O método construtor cria para o objeto Diretório um canal de comunicação com o endereço especificado em tempo de compilação. O método destrutor fecha o canal de comunicação aberto, removendo o processo que executa o cluster do objeto Diretório.

A estrutura do método responsável pela criação dos objetos distribuídos é apresentada na figura 5.6. A primeira tarefa envolvida é identificar, através dos parâmetros recebidos, a classe distribuída da qual o objeto deve ser instanciado. A seguir é criado o processo para executar o cluster que a implementa. O método aguarda o recebimento de uma mensagem informando o endereço da interface de acesso do objeto criado, enviando os parâmetros rece-

bidos para o método construtor da classe. O endereço de localização do objeto criado é enviado ao objeto que requisitou a criação.

```
Diretorio::CriaObjeto ( int idclasse, msginvocacao parametros )
{
 <cria cluster para executar objeto da classe "idclasse">
 <recebe endereço do objeto criado>
 <envia "parametros" como invocação ao método construtor da classe>
 <envia endereço do objeto criado para o objeto requisitante>
}
```

Figura 5.6: Algoritmo de criação de objetos distribuídos

5.3 A Aplicação Implementada

A linguagem protótipo DPC++ foi utilizada para implementar um algoritmo de geração de fractais. Com a execução da aplicação obtida foi possível analisar o desempenho propiciado pela utilização do modelo.

5.3.1 Fractais de Mandelbrot

A teoria de fractais é utilizada na modelagem de árvores, nuvens, texturas e outras formas não lineares na geração de cenas de alto grau de realismo [OSO89]. Os algoritmos de geração de fractais em geral requerem grande carga de processamento. Alguns destes algoritmos podem ser implementados de forma concorrente possibilitando diminuição do tempo necessário para geração de uma imagem. Este é o caso dos algoritmos fractais de Mandelbrot.

Os fractais de Mandelbrot pertencem a uma família de fractais gerados a partir de funções complexas, utilizando uma função de números complexos para calcular a apresentação dos pontos que compõem uma cena. O algoritmo de Mandelbrot é apresentado na figura 5.7, neste algoritmo verifica-se que os pontos podem ser calculados de forma independente. A carga computacional deste

algoritmo é equivalente ao número de iterações necessárias para obter o valor de um ponto.

```
Para cada ponto do universo fractal

  Calcula  Z' = Z * Z + Z
  Enquanto módulo(Z) < 2 OU  nro. iterações menor que limite de iterações

  Se nro. iterações realizadas maior OU igual ao limite de iterações
    Então Não divergiu
          Exibe ponto
```

Figura 5.7: Algoritmo de Mandelbrot

5.3.2 Modelo da aplicação

Para implementar de forma concorrente o algoritmo de Mandelbrot foi aproveitada a característica de independência dos pontos. A área total a ser calculada foi dividida em regiões, calculadas em separado. O conjunto de regiões calculadas compõem a área desejada do fractal.

O programa consiste em uma função de inicialização do programa, responsável por instanciar o primeiro objeto e na interação entre objetos de três classes distribuídas: *Distribuição*, *Cálculo* e *Saída*. A figura 5.8 apresenta a forma de interação entre os objetos destas classes.

O primeiro objeto instanciado pertence a classe *Distribuição*. Seu método construtor cria um objeto da classe *Saída* e n objetos da classe *Cálculo*. O objeto *Distribuição* controla a distribuição das regiões a serem calculadas pelos objetos de *Cálculo* através de um único método, o qual, quando invocado, retorna os limites da região a ser calculada.

Os objetos da classe *Cálculo* são responsáveis por processar o cálculo de regiões fractais. As regiões são solicitadas ao objeto *Distribuição* através de mensagens síncronas. Uma vez calculada a região, os resultados são enviados

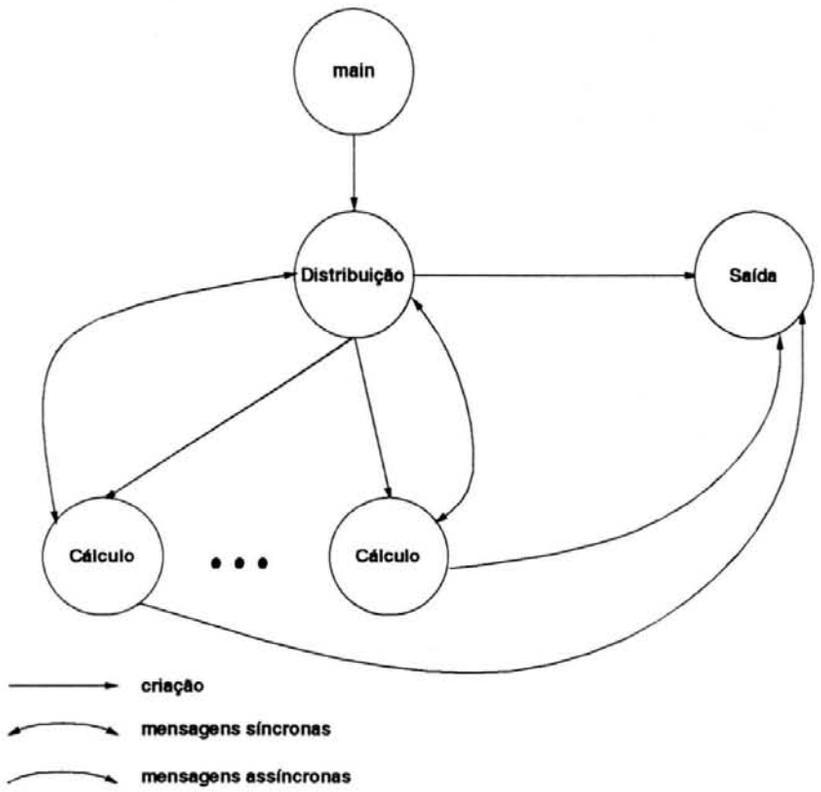

Figura 5.8: Modelo da aplicação

através de uma mensagem assíncrona ao objeto *Saída*. O único método do objeto de cálculo é responsável pelo processamento das regiões após receber uma mensagem assíncrona indicando o início do processamento.

O número de regiões processadas de forma concorrente depende do número de objetos da classe *Cálculo* criados. Na implementação realizada são criados 4 objetos de cálculo.

Ao contrário dos demais objetos, que podem ser criados em qualquer nodo processador, o objeto *Saída* requer uma estação com vídeo gráfico policromático. O objeto *Saída* possui um método que recebe mensagens assíncronas contendo uma região do fractal a ser exibida.

Nos anexos é apresentado o código parcial que implementa a aplicação apresentada, ilustrando sua implementação. O anexo A-4 apresenta o código que implementa a classe *Diretório* e o anexo A-3 a classe *Cálculo*.

5.3.3 Desempenho obtido

O ambiente em que foram realizadas as medições para análise do desempenho da aplicação conta com 4 estações de trabalho interconectadas por uma rede Ethernet de 10 Mbits. A esta rede encontram-se ligadas outras estações não utilizadas pela aplicação no experimento. São utilizadas estações Sun modelos SLC e Sparc-2. O desempenho nominal destes equipamentos é de 17 MIPS-SPEC e 2 MFLOPS para o modelo SLC e 29 MIPS e 4 MFLOPS para o modelo Sparc-2. Este ambiente foi utilizado no experimento de forma compartilhada com os demais usuários e aplicativos da rede.

Foram realizadas medições de tempo para geração de uma área fractal específica variando o número de iterações para o cálculo dos pontos e o número de estações utilizadas. A área do fractal é apresentada na figura 5.9, tendo sido calculada com 5.000 e 10.000 iterações por ponto em 1, 2 e 4 estações de trabalho. Outras medições de desempenho do protótipo podem ser encontradas em [CAV93c].

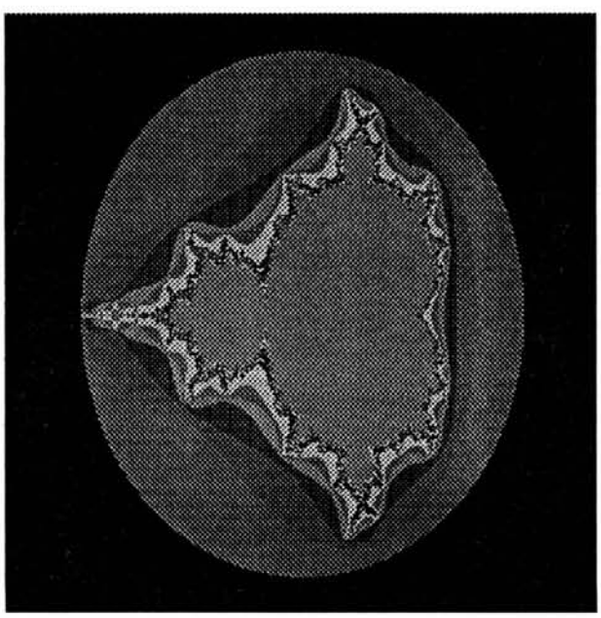

Figura 5.9: Área do fractal utilizada no experimento

Os índices de desempenho colhidos na execução da aplicação encontram-se na tabela 5.1. São apresentados os tempos cronometrados em cada caso e o *speed-up* da execução em referência à execução em uma única estação.

Tabela 5.1: Desempenho obtido na aplicação

5.000 iterações			**10.000 iterações**		
N. estações	**Tempo**	**Speed-up**	**N. estações**	**Tempo**	**Speed-up**
1	20′ 02″	–	1	41′ 55″	–
2	15′ 28″	1,3	2	30′ 09″	1,4
4	8′ 21″	2,4	4	19′ 09″	2,2

6 CONCLUSÃO

Obter ganho de desempenho na execução de programas, alcançar bons níveis de integridade de dados e programar sistemas de forma modular são objetivos desejáveis na construção de qualquer aplicação computacional. Estes objetivos definem, respectivamente, o tempo de resposta, a segurança de dados e a clareza no estilo de programação. Para dispor destas qualidades, os projetistas de software buscam ferramentas de programação (linguagens) cada vez mais poderosas e eficientes, tanto no desenvolvimento quanto na execução de aplicações.

Diversos paradigmas tem sido desenvolvidos para linguagens a fim de suprir estas necessidades. O paradigma de **programação orientada a objetos** [TAK90] é um exemplo de busca de simplicidade de escrita e compreensão de programas, além de oferecer estruturas de encapsulamento de dados, favorecendo a escrita de grandes sistemas. Já os diversos modelos de **programação concorrente** [BAL89] visam melhorar o desempenho na execução de programas, porém, por possuírem estruturas mais complexas, não são comumente utilizados em sistemas de grande porte. A união destes dois paradigmas provê os recursos desejados pelos projetistas de software.

Apesar de trazer muitas vantagens ao projeto, implementação e manuntenção de sistemas [SNY86], a programação orientada a objetos, em arquitetura de computadores convencionais, apresentam baixo desempenho e ineficiência no uso de recursos. Segundo [HUF89] isto se deve aos níveis de suporte utilizados, tanto de software básico como de hardware, não serem os mais apropriados às linguagens orientadas a objetos.

Devido a sua semelhança com sistemas paralelos de memória distribuída, principalmente por possuirem o mesmo mecanismo de comunicação ba-

seado em mensagens, a programação orientada a objetos não precisa ser sinônimo de ineficiência [ASS92]. Sendo implementada com técnicas de paralelismo, linguagens orientadas a objetos permitem explorar os benefícios da programação possibilitando maior desempenho no processamento de aplicações.

Confirmando este pensamento, [YAU92] afirma que, ao ser comparado com modelos orientados a fluxo de dados ou a comunicação, o modelo orientado a objetos mostra-se como o mais promissor no desenvolvimento de sistemas distribuídos. Como resultado, verifica-se um crescente interesse no uso de técnicas orientadas a objetos como base para soluções concorrentes de uma vasta gama de problemas [HOP89].

Na introdução do paralelismo a linguagens orientadas a objetos a bibliografia apresenta duas linhas de pesquisa [AUG92]: utilização de paralelismo explícito e implícito. Na linha que explora o paralelismo explícito encontra-se o maior número de trabalhos, onde são abordadas construções que tornam possível o processamento concorrente de linguagens orientadas a objetos. Já nos trabalhos que exploram o paralelismo implícito, a tese defendida é que a partir de uma linguagem orientada a objetos é possível a geração automática de programas paralelos.

Exemplos de linguagens com paralelismo explícito podem ser encontradas em [WYA92, AUG92]. Em [YIN90] há um exemplo de aplicação de paralelismo implícito sobre uma linguagem orientada a objetos.

Neste trabalho foi apresentado um modelo para exploração da execução distribuída de aplicações pelo uso de uma linguagem orientada a objetos. Como resultado da aplicação do modelo, é oferecida uma linguagem distribuída orientada a objetos, fornecendo desempenho otimizado do programa com simplicidade de escrita do código, voltada a aplicações com paralelismo de dados com granulosidade larga a ser implementado sobre uma rede de múltiplos processadores independentes totalmente conectados.

6.1 O Modelo

O modelo proposto foi desenvolvido para suportar o processamento distribuído de uma linguagem orientada a objetos em uma arquitetura multicomputadora. O modelo necessita uma configuração de hardware que disponha de nodos de processamento com capacidade de multiprocessamento e uma rede de comunicação capaz de permitir o envio de mensagens entre todos os nodos processadores. A apresentação do modelo ocorreu com a discussão de seus dois níveis: o *nível de linguagem* e o *nível operacional*.

A estrutura da programação em uma linguagem que implementa o modelo é apresentada no nível de linguagem. Neste nível são apresentadas restrições no uso de memória compartilhada e a sintaxe mínima necessária ao modelo. Sendo implementado em um ambiente distribuído, o modelo não possui uma área de memória comum, portanto não é possível qualquer compartilhamento de dados ou código entre os objetos. A herança, cuja principal característica é o compartilhamento de código entre classes de objetos, no modelo apresentado é implementada através de cópia.

A sintaxe mínima definida para a linguagem distribuída consiste em proporcionar meios do programador diferenciar as classes de objetos distribuídos das classes de objetos locais e de especificar a forma de interação entre os métodos de classes distribuídas. O modelo prevê que a interação entre os métodos possa ser através de invocações síncronas, assíncronas ou ainda assíncrona com confirmação. As demais características da linguagem possuem implementações semelhantes às linguagens orientadas a objetos seqüenciais.

O nível operacional implementa os recursos de suporte à execução da linguagem distribuída. É provido um mecanismo de controle, o Diretório, onde as informações da execução da aplicação são mantidas. Também neste nível, encontram-se os recursos para criação de objetos e os mecanismos de suporte a

troca de mensagens utilizados pelo nível de linguagem. O modelo implementa um esquema de *objetos procuradores* permitindo que o acesso a objetos remotos seja realizado com a mesma semântica dos acessos a objetos locais.

O uso de um *compilador* possibilita a transparência do acesso aos recursos do ambiente distribuído provido pelo nível operacional. Além de verificar a sintaxe do programa, o compilador introduz os elementos pertencentes ao nível operacional no programa escrito na linguagem que implementa o nível de linguagem do modelo.

6.2 A Linguagem DPC++

Redes locais de computadores estão cada vez mais disseminadas, sendo sua utilização uma alternativa aos dispendiosos *mainframes* ou mesmo supercomputadores. O uso destas redes encontra atualmente um obstáculo na área de programação. São poucas as ferramentas que tiram proveito dos recursos de distribuição, estando estas no estado da arte do desenvolvimento na área. Neste trabalho foi apresentada a linguagem DPC++, *Processamento Distribuído em* C++, que consiste em uma ferramenta de programação que implementa o modelo distribuído proposto, estando direcionada para proporcionar a programação em um ambiente de rede com transparência aos recursos distribuídos.

DPC++ é uma linguagem que reúne os paradigmas de programação orientada a objetos e de processamento distribuído. Sua utilização permite acesso aos recursos do processamento distribuído de uma forma transparente, cabendo ao programador modelar a aplicação segundo o modelo de orientação a objetos. O uso de um compilador responsável por introduzir o modelo distribuído em um programa submetido garante este nível de transparência.

A linguagem DPC++ foi desenvolvida a partir de C++, da qual herda a sintaxe e toda a estrutura de programação. Alguns recursos de C++ não puderam ser implementados em DPC++, devido a troca de ambiente com memória centralizada por um ambiente com memória distribuída. O suporte ao processamento de protótipo DPC++ é provido por uma rede de estações Sun e pelo sistema operacional SunOS.

Sendo baseada em C++, DPC++ possibilita a reutilização de códigos já escritos, também facilitando o acesso aos programadores que conhecem a linguagem base. Introduzindo a nível de linguagem poucas estruturas sintáticas, programas DPC++ podem ser desenvolvidos e testados em ambiente seqüencial, sendo posteriormente convertidos para o ambiente distribuído. Este recurso possibilita obter programas mais confiáveis, pois a tarefa de depuração em ambientes concorrentes é bastante complexa.

De acordo com o modelo que suporta a execução distribuída, DPC++ é uma linguagem distribuída voltada a aplicações com granulosidade de processamento médias ou grossas. Sendo as unidades de processamento os objetos distribuídos, a adequação do grão de processamento em DPC++ é realizado pelo programador na modelagem das classes distribuídas. Desta forma, é simplificada a expressão do paralelismo, uma vez que um objeto já representa um elemento autônomo e comunicando com outros objetos através de mensagens.

A granulosidade suportada pelo modelo está adequada a implementação em redes de estações de trabalho equivalentes a utilizada no desenvolvimento do protótipo. O custo da comunicação nestes ambientes é bastante alto, portanto grânulos grossos de processamento mostram-se os que geram uma maior eficiência no uso dos recursos [LOU92], diminuindo a taxa de comunicação e aumentando a carga de processamento.

ANEXO A-1 A LINGUAGEM C++

Neste anexo é apresentada um rápida visão das características da linguagem de programação C++. Uma visão detalhada pode ser encontrada em [TAK90, STR86, ELL90].

A linguagem C++ [ELL90] foi desenvolvida pelos laboratorios Bell AT & T, sendo derivada do C [KER78]. Como a linguagem original, C++ é voltada a programação de prósito geral [STR86]. C é totalmente compatível com C++ [BOR90], possuindo extenções que suportam abstração de dados e programação orientada a objetos. Um conjunto de regras para a programação em C++ é apresentado em [NYQ92].

A maioria dos recursos utilizados em C++ form herdados de C, tais como tipos básico de dados, operação e sintaxe de instruções e estrutura do programa. Os recursos acrescentados, como classes, abstração de dados, herança e polimorfismo, suportam novas técnicas de programação ao mesmo tempo que aprimoram as partes parecidas com C.

O uso de classes em C++ permite que tipos de dados agregados (tipo estrutura) sejam definidos junto com funções que operam sobre estes dados. Tipos de dados definidos em uma classe são internos a ela, provendo abstração de dados. O mecanismo de herança de classes estende a abstração de dados até a programação orientada a objetos.

Os tipos de dados existentes em C, ao serem utilizados em C++, não são convertidos para classes. Desta forma, classes C++ convivem com os tipos padrões de C.

Além dos recursos que suportam técnicas para a construção de estruturas de dados, há também o mecanismo de "sobrecarregamento de funções",

chamado de polimorfismo ou *overloading*. O polimorfismo possibilita que duas ou mais funções tenham o mesmo nome, conquanto diferenciem-se no tipo de argumentos que recebem.

A compilação de um programa C++ geralmente necessita de um compilador C++ e um compilador C. O compilador C++ converte o programa C++ em um programa C, o qual é submetido ao compilador C, gerando o código executável.

No restante deste capítulo, serão apresentados os recursos de C++ que permitem a programação orientada a objetos. Serão considerados os itens relevantes a implementação de um modelo de distribuição sobre esta linguagem.

A-1.1 Classes em C++

Uma classe C++ descreve, em termos de atributos e operadores associados, as características de um conjunto de objetos. A descrição, denominada definição, de uma classe é dividida em duas partes: (*I*) sua especificação, contendo as classes de níveis hierárquicos superiores, se existirem, resultante do processo de herança, os seus atributos e seus protótipos de métodos; e (*II*) sua implementação, onde encontra-se a implementação dos métodos da classe. Uma especificação de classe é muito semelhante a especificação de tipos em C, assim como a implementação de métodos é semelhante a implementação de funções.

À instanciação de variáveis a partir de tipos definidos de classes se dá da mesma forma que a instanciação de variáveis de outros tipos quaisquer. A instanciação, neste caso, implica na criação de variáveis às quais denominamos objetos. Objetos podem ser criados de forma estática ou dinâmica, neste segundo caso, o objeto é acessado através de um ponteiro.

A-1.1.1 Membros de classes

Uma classe C++ é montada encapsulando atributos e métodos. Os atributos correspondem a variáveis definidas a partir de tipos abstratos de dados, como inteiros (*int*), caracter (*char*) ou tipos de dados complexo do tipo estrutura (*struct*). Os métodos correspondem às funções implementadas pela classe.

No momento de instanciação de um objeto a partir de uma classe, é alocado espaço para o objeto. Também instancias dos membros de dados são criadas e inicializadas como componentes do objeto.

Os métodos são funções declaradas no interior de uma classe. Os métodos executam operações definitas para classe, podendo acessar os componentes tipo atributo internos a classe que pertencem. O código que implementa os métodos pode estar definido sob forma de função, fora do escopo de definição da classe, ou *inline*, dentro do escopo de definição da classe, imediatamente a seguir da definição do método.

Os métodos *inline* tem uma estrutura similar a definição de macros de C definidas através da diretiva *#define*. O compilador C++ substitui as chamadas aos métodos *inline* pelo código que o implementa. Normalmente o compilador avalia a complexidade do método verificando se é justificada a implementação *inline*. Caso considere que não seja, o método é automaticamente convertido para um método comum. Logo, a opção por definir um método *inline* é viável quando seu código for simples e curto, não justificando o gasto extra com uma chamada e retorno de função.

A-1.1.2 Acesso aos membros

O acesso aos componentes de um objeto instanciado se dá pelos operadores . (ponto) e -> (flecha), independente se os componentes são dados ou funções. O operador . é utilizado quando se tem um objeto de uma classe e -> quando se tem um ponteiro para objeto.

A-1.1.3 Níveis de visibilidade

As classes seguem as regras de escopo de C, bem como os objetos criados a partir delas. Portanto, objetos cujo escopo seja todo um arquivo ou sendo o objeto do tipo *static*, sua criação ocorre antes do inicio da execução do programa e sua vida dura toda a execução. Ao contrário, objetos locais, bem como parâmetros recebidos por funções tem vida efemera, sendo criados no momento em que a execução do programa atinge sua declaração e removidos ao final do bloco onde foram declarados.

Caso a criação do objeto seja dinâmica, através do comando *new* que retorna o endereço de memória do objeto criado, a vida do objeto também é durante toda a execução do programa ou até serem explicitamente removidos com o comando *delete*.

O acesso aos membros internos de classes são regidas por 3 níveis de visibilidade:

private : o acesso aos membros declarados com este nível de visibilidade são acessados somente pelo próprio objeto criado. Uma classe também pode permitir o acesso aos seus dados privados a outras classes ou a funções isoladas, declarando-as como "amigas" (*friend*).

public define o protocolo de manipulação das instâncias de uma classe, podendo ser acessados por qualquer parte do programa.

protected : consiste em um nível intermediário de visibilidade, estendendo às subclasses a possibilidade de acesso ao conjunto de membros com este atributo.

A-1.2 Compartilhamento de Memória

C++ permite dois tipos de compartilhamento de memória: compartilhamento de código e compartilhamento de dados. O compartilhamento de código é efetuado através de métodos *in-line* e do mecanismo de herança. O de métodos *in-line* implicam não na chamada de uma função, mas na duplicação de seu código em tempo de compilação. O compartilhamento de código também ocorre quando da utilização do mecanismo de herança, onde as implementações dos métodos das classes herdadas são utilizadas pelas sub-classes.

O compartilhamento de dados é permitido através do uso de variáveis de classe e da passagem por parâmetros. As variáveis de classe consistem em células únicas de memória pertencente a uma classe. Todas as instâncias desta classe compartilham o uso desta variável.

Além da passagem de parâmetros por valor, C++ permite que parâmetros sejam passados a métodos por referência ou por endereço. A passagem de parâmetros por referência implica na possibilidade de alteração da posição de memória ocupada pelo objeto. A passagem de parâmetros por endereço de memória é uma característica herdada de C, cujo uso no modelo de objetos não é recomendado, uma vez que fere o princípio de encapsulamento de informação por possibilitar acesso direto aos dados armazenados em memória.

ANEXO A-2 CLASSE COMUNICAÇÃO

Neste anexo encontra-se a implementação da classe utilizada para prover a implementação do *servidor de mensagens* do modelo. A implementação conta com a estruturação dos serviços sob a forma da classe *Comunicacao*.

```
class Comunicacao
{
    private:
          char  mensagem[MSG_LEN];
          void CriaSocket(int);

    public:
          int sock;
         /* numero do socket usado na comunicacao           */
         struct sockaddr_in from;
         /* endereco do emissor da ultima mensagem recebida */
         struct sockaddr_in myaddr;
         /* endereco para comunicacao do proprio objeto     */
         SOCK_ADDR addr;

         Comunicacao();
         Comunicacao(int);
         ~Comunicacao();
         void Envia       ( char * ,int, struct sockaddr_in );
         char * RecebeDad ( void );
         char * RecebeCon ( void );
         void RespondeDad ( char *, int );
         void RespondeCon ( void );
};

class CtrlMsg : public Comunicacao
{

     public:
       void EnviaAss     ( char *, int, struct sockaddr_in  );
       char * EnviaSin   ( char *, int, struct sockaddr_in  );
       void EnviaConf    ( char *, int, struct sockaddr_in  );

       char * RecebeAss  ( void );
       char * RecebeSin  ( char *, int );
       char * RecebeConf ( void );

};


EXTERN_FUNCTION( extern int close,(int) );

/*************************************************************/
/*  Metodos Contrutores e Destrutores da classe Comunicacao*/
/*************************************************************/
/*************************************************************/
/* METODO     : Contrutor                                    */
/* Descricao  : cria socket para datagrama e informa numero*/
/*              do socketcriado                              */
/*                                                           */
/* Autor      : Gerson Cavalheiro                            */
```

```
/*                                                                  */
/* Entrada     : nenhuma                                            */
/*                                                                  */
/* Saida       : numero do socket criado                            */
/*                                                                  */
/********************************************************************/
Comunicacao::Comunicacao (int porta)
{
   CriaSocket(porta);
}

Comunicacao::Comunicacao(void)
{
  CriaSocket(0);
}


/********************************************************************/
/* METODO      :   Destrutor                                        */
/* Descricao   :   Encerra comunicacao fechando o socket            */
/*                 criado pelo construtor                           */
/* Entrada     :   nenhuma                                          */
/*                                                                  */
/* Saida       :   nenhuma                                          */
/*                                                                  */
/********************************************************************/

Comunicacao::~Comunicacao()
{
 close(sock);
}


/********************************************************************/
/* METODO      :   CriaSocket                                       */
/* Descricao   :   Encerra comunicacao fechando o socket            */
/*                 criado pelo construtor                           */
/* Entrada     :   nenhuma                                          */
/*                                                                  */
/* Saida       :   nenhuma                                          */
/*                                                                  */
/********************************************************************/

void Comunicacao::CriaSocket(int porta)
{
 int                length;

/*** CRIA SOCKET **********************/
 if ( (sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) {
  perror("Opening stream socket");
  exit(1);
 }

/*** FAZ BIND *****************************/
 myaddr.sin_family = AF_INET;
 myaddr.sin_addr.s_addr = INADDR_ANY;
 myaddr.sin_port = porta;/*se porta == zero cria um socket*/

/* bind associa um nome a um socket */
 if (bind(sock,(struct sockaddr *)&myaddr,sizeof(myaddr)) < 0)
{
     perror("Binding stream socket");
     exit(1);
 }

 length = sizeof(myaddr);
 if (getsockname(sock,(struct sockaddr *)&myaddr,&length) < 0)
{
     perror("getting socket name");
```

```
    exit(1);
 }

 addr.porta=getporta(myaddr);
 addr.maquina=getmaquina();
 addr.addr=myaddr;
}

/*************************************************************/
/*  Metodos Publicos da classe Comunicacao                   */
/*************************************************************/
/*************************************************************/
/* METODO     :   Envia                                      */
/* Descricao  :   Envia mensagem,via datagrama,para destino*/
/*                informado com tamanho de 512 Kb Maximo    */
/* Entrada    :   msg  - mensagem a ser enviada              */
/*                tam  - tamanho da mensagem a ser enviada */
/*                to   - estrutura que identifica o destino*/
/*                                                          */
/* Saida      :   nenhuma                                    */
/*                                                          */
/*************************************************************/
void Comunicacao::Envia(char *msg,int tam,struct sockaddr_in
to)
{
    if ( tam > MSG_LEN )
    {
printf("Mensagem nao pode ultrapassar %d bytes\n",MSG_LEN);
printf("Mensagem nao enviada\n");
return;
    }

    if ( sendto(sock , msg , tam ,0,
       (struct sockaddr *) &to,sizeof(to)) < 0 ) {
perror("sending datagram message  ");
exit(1);
    }
} /* envia */

/*************************************************************/
/* METODO     : RespondeDad                                  */
/* Descricao  : Envia mensagem de resposta para o ultimo     */
/*              endereco que enviou mensagem para            */
/*              "Recebe"->from                               */
/* Entrada    : msg - ponteiro para mensagem de resposta     */
/*                                                          */
/* Saida      : nenhuma                                      */
/*                                                          */
/*************************************************************/
void Comunicacao::RespondeDad( char *msg, int tam )
{
 Envia(msg,tam,from);
}
/*************************************************************/
/* METODO     : RespondeCon                                  */
/* Descricao  : Envia resposta de CONFirmacao para o ultimo */
/*              endereco que enviou mensagem para            */
/*              "Recebe"->from                               */
/* Entrada    : nenhuma                                      */
/*                                                          */
/* Saida      : nenhuma                                      */
/*                                                          */
/*************************************************************/
void Comunicacao::RespondeCon( )
{
   Envia(CONF,CONF_LEN,from);
}

/*************************************************************/
```

```
/* METODO      : RecebeDad                                          */
/* Descricao   : Recebe mensagem, via datagrama, com tamanho */
/*               maximo de 512 Kb (MSG_LEN), alocando espaco */
/*               exato para amensagem recebida e retornando  */
/*               seu endereco para o "chamador"              */
/* Entrada     :                                              */
/*                                                            */
/* Saida       : msg - ponteiro para endereco de onde deve   */
/*                     ser colocada a mensagem recebida       */
/*                                                            */
/************************************************************/
char * Comunicacao::RecebeDad ( void )
{
  int flen = sizeof(struct sockaddr_in) ;

 if ( recvfrom(sock,(char *) mensagem,MSG_LEN,0,
      (struct sockaddr *) &from, (int *) &flen) < 0 )
  {
      perror("receiving datagram message");
      exit(1);
  }

  return(mensagem);

} /* RecebeDad */

/************************************************************/
/* METODO      : RecebeCon                                    */
/* Descricao   : Recebe resposta de CONFirmacao, via          */

/*               datagrama, com tamanho maximo de 512 Kb,     */

/*               alocando, espaco exato para a mensagem       */
/*               recebida eretornando seu endereco para o     */
/*               "chamador"                                   */
/* Entrada     : nenhuma                                      */
/*                                                            */
/* Saida       : msg - ponteiro para endereco de onde deve   */
/*                     ser colocada a mensagem recebida       */
/*                                                            */
/************************************************************/
char * Comunicacao::RecebeCon( void )
{
  int flen = sizeof(struct sockaddr_in) ;
  char CONFirma[2];

  if ( recvfrom(sock,(char *) CONFirma,CONF_LEN,0,
       (struct sockaddr *) &from, (int *) &flen) < 0  )
  {
      perror("receiving datagram message");
      exit(1);
  }

   return(CONFirma);

} /* RecebeCon */


/************************************************************/
/*  Metodos Publicos da classe CtrlMsg                      */
/************************************************************/
/************************************************************/
/* METODO      : EnviaAss                                     */
/*                                                            */

/* Descricao   : Envia mensagem assincrona                    */
/*                                                            */
/* Entrada     : msg - ponteiro para mensagem a ser enviada*/
```

```
/*              tam - tamanho da mensagem a ser enviada   */
/*               to - estrutura que identifica o destino  */

/*                                                        */
/* Saida      : nenhuma                                   */
/*                                                        */
/**********************************************************/
void CtrlMsg::EnviaAss(char * msg, int tam, struct sockaddr_in
to)
{
   Envia(msg,tam,to);
}

/**********************************************************/
/* METODO     : EnviaSin                                  */
/*                                                        */

/* Descricao  : Envia mensagem sincrona, espera Dados de  */
/*              resposta                                  */
/* Entrada    : msg  - ponteiro para mensagem a ser enviada*/
/*              to   - estrutura que identifica o destino */

/*                                                        */
/* Saida      : resp - mensagem com dados de resposta     */
/*                                                        */
/**********************************************************/
char * CtrlMsg::EnviaSin(char * msg, int tam, struct sockaddr_in
to)
{
    char * resp;

    Envia(msg,tam,to);

    resp = RecebeDad();

    return(resp);

}

/**********************************************************/
/* METODO     : EnviaConf                                 */
/*                                                        */

/* Descricao  : Envia mensagem sincrona, espera resposta de*/
/*              CONFirmacao                               */
/*                                                        */
/* Entrada    : msg  - ponteiro para mensagem a ser enviada*/
/*              tam  - tamanho da mensagem a ser enviada  */
/*               to  - estrutura que identifica o destino */

/*                                                        */
/* Saida      : nenhuma                                   */
/*                                                        */
/**********************************************************/
void CtrlMsg::EnviaConf(char * msg, int tam, struct sockaddr_in
to)
{
   char * CONFirma;

   Envia(msg,tam,to);

   CONFirma =  RecebeCon();

}

/**********************************************************/
/* METODO     : RecebeAss                                 */
/*                                                        */
```

```
/* Descricao  : Recebe Mensagem assincrona                  */
/*                                                          */
/* Entrada    : nenhuma                                     */

/*                                                          */
/* Saida      : msg - ponteiro para mensagem recebida       */
/*                                                          */
/************************************************************/
char * CtrlMsg::RecebeAss ( void )
{
   char * msg;

   msg = RecebeDad();

   return(msg);

}
/************************************************************/
/* METODO     : RecebeSin                                   */
/*                                                          */

/* Descricao  : Recebe Mensagem sincrona e envia de volta   */
/*              dados de resposta                           */
/*                                                          */
/* Entrada    : dad - dados de resposta                     */
/*              tam - tamanho dos dados de resposta         */

/*                                                          */
/* Saida      : msg - ponteiro para mensagem recebida       */
/*                                                          */
/************************************************************/
char * CtrlMsg::RecebeSin( char * msgresp, int tam)
{
   char * msg;

   msg = RecebeDad();

   RespondeDad(msgresp,tam);

   return(msg);
}

/************************************************************/
/* METODO     : RecebeCon                                   */
/*                                                          */

/* Descricao  : Recebe Mensagem sincrona e envia de volta   */
/*              sinal de CONFirmacao                        */
/*                                                          */
/* Entrada    : nenhuma                                     */

/*                                                          */
/* Saida      : msg - ponteiro para mensagem recebida       */
/*                                                          */
/************************************************************/
char * CtrlMsg::RecebeConf( void )
{
   char * msg;

   msg = RecebeDad();

   RespondeCon();

   return(msg);
}
```

ANEXO A-3 A CLASSE CÁLCULO

Neste anexo é apresentado o cluster da classe *Cálculo* e a classe procuradora para objetos desta classe. Com estes exemplos espera-se dar uma visão da implementação de objeto distribuídos e da tarefa a ser realizada pelo compilador DPC++.

A-3.1 O Cluster de Cálculo

Abaixo encontra-se a listagem parcial da implementação do cluster que suporta a execução de objetos da classe *Cálculo*.

```
/*---------------------DEFINICAO DA CLASSE DE CALCULO----------------------*/

class CALCULO {

Ponto         * pontos;
Coord_Parcial Area_Parcial;
DISTRIBUICAO_P     * addr_dist;
             SAIDA_P            * addr_saida;


void CalculaArea(void);

public:
CALCULO( DISTRIBUICAO_P *, SAIDA_P * );
                void Calcula(void);
~CALCULO( );

};

/*-------------------implementacao da classe de CALCULO--------------------*/

CALCULO::CALCULO ( DISTRIBUICAO_P * ad, SAIDA_P * as )
{

   addr_dist  = ad;
   addr_saida = as;
}


CALCULO::~CALCULO()
{
   free((char*)pontos);
}
```

```
void CALCULO::Calcula(void)
{
   Area_Parcial = addr_dist->MontaArea();

   pontos = (Ponto *)malloc(sizeof(Ponto) * (Area_Parcial.npx*Area_Parcial.npy)
);

   while ( Area_Parcial.it != 0 ) {

CalculaArea();

addr_saida->PutArea( (Ponto *)pontos, Area_Parcial.npx, Area_Parcial.npy
);

Area_Parcial = addr_dist->MontaArea();

free((char*)pontos);

pontos = (Ponto *)malloc(sizeof(Ponto) * (Area_Parcial.npx*Area_Parcial.npy)
);

   }
}


void CALCULO::CalculaArea()
{
  double ac, bc , a, b, b1, size;
  int x, y, cont;

  bc    = Area_Parcial.iniy + Area_Parcial.npy * Area_Parcial.vary;

  for ( y = Area_Parcial.npy ; y > 0 ; --y )  {
ac = Area_Parcial.inix;
for ( x = 0 ; x < Area_Parcial.npx ; ++x )
{
/* calcula z=zÃ2+c */
     a    = ac;
     b    = bc;
     size = 0.0;
     cont = 0;
     while( (size < 4.0) && (cont < Area_Parcial.it) )
     {
   b1   = (double)2.0 * a * b;
   a    = (double)a*a - (double)b*b + (double)ac;
   b    = (double)b1 + (double)bc;
   size = (double)a*a + (double)b*b;
     ++cont;
     }
     ac += Area_Parcial.varx;

     pontos[ (y*Area_Parcial.npx) + x ].x   = (Area_Parcial.pix
+ x);
     pontos[ (y*Area_Parcial.npx) + x ].y   = (Area_Parcial.piy
+ y);
     pontos[ (y*Area_Parcial.npx) + x ].cor = cont;

}
bc -= Area_Parcial.vary;
  }

}


void main (int argc, char **argv)
{
   CALCULO * calc;

   DIRETORIO_P * dirp;
```

```
   CtrlMsg     ServMsg;
   char        mensagem[MSG_LEN];
   int         * metodo;

   SOCK_ADDR ad,as;      /* para receber parametros */

   dirp = new DIRETORIO_P(2222,"mate");


   sprintf(mensagem,"%d %s",ServMsg.addr.porta,ServMsg.addr.maquina);
   /* Envia endereco para o diretorio */

   ServMsg.Envia(mensagem,strlen(mensagem),dirp->COM.myaddr);


   while ( TRUE ) {
            mensagem = ServMsg.ReceDad();

            switch ( Metodo(mensagem) ) {
                  case M_CALCULO_1 :

                                    DISTRIBUICAO_P * distp;
                                    SAIDA_P        * saidap;

                                     sscanf(mensagem,"%d %s %d
%s",&ad.porta,
                                            ad.maquina,&as.porta,as.maquina);

                                     distp  = new DISTRIBUICAO_P(ad.porta,ad.maquina);
                                     saidap = new SAIDA_P(as.porta,as.maquina);

                                     calc = new CALCULO(distp,saidap);
                                     break;
                  case M_CALCULO_2 :
                                     calc->Calcula();
                                     break;
                  case M_CALCULO_3 :
                                     delete ( calc );
                                     exit ( 0 );
            }
   }
}
```

A-3.2 Classe Procuradora de Cálculo

A classe que implementa objetos procuradores para o objeto cálculo encontra-se listada abaixo.

```
/*---------------------------CLASSE PROCURADORA DE CALCULO--------------------*/
class CALCULO_P
{
               Ponto         * pontos;
               Coord_Parcial Area_Parcial;
               DISTRIBUICAO_P * addr_dist;
```

```
SAIDA_P         * addr_saida;

                char parametros[MSG_LEN];
                CtrlMsg COM;

        public:

                SOCK_ADDR objD;

                DIRETORIO_P * dirp;

CALCULO_P   (DISTRIBUICAO_P *, SAIDA_P *);
                void Calcula_P (void);
                CALCULO_P   (int, char*);
                ~CALCULO_P (void);


};

/*-------------------definicao da classe procuradora de CALCULO-----------*/


/* Metodo construtor:
   - cria procurador para Diretorio
   - cria procurador para distribuicao e saida
   - invoca no Diretorio a criacao de objeto       */

CALCULO_P::CALCULO_P(DISTRIBUICAO_P * ad, SAIDAP_P * as)
{
   dirp = new DIRETORIO_P(2222,"mate");

   sprintf(parametros,"%d %s %d %s",ad.objD.porta,ad.objD.maquina,
           as.objD.porta,as.objD.maquina);

   objD = dirp->CriaObjeto(CALCULO,parametros,strlen(parametros));
}


/* Metodo construtor:
   - cria procurador para Diretorio
   - seta endereco do servidor de mensagens para o objeto distribuido
*/

CALCULO_P::CALCULO_P(int porta, char * maquina)
{
   dirp = new DIRETORIO_P(2222,"mate");

   sprintf(parametros,"%d %d %s %d %s",M_CALCULO_1,ad.objD.porta,ad.objD.maquina,
           as.objD.porta,as.objD.maquina);

   objD = setaddr(porta,maquina);
}


/* Metodo responsavel pelo inicio do calculo */
CALCULO_P::Calcula_P ( void )
{
 sprintf(parametros,"%d",M_CALCULO_2);
 COM.EnviaAss(parametros,sizeof(int),objD.addr);
}


/* Metodo destrutor */
CALCULO_P::~CALCULO_P()
{
   sprintf(parametros,"%d",M_CALCULO_3);
   COM.EnviaAss(parametros,sizeof(int),objD.addr);
}
```

ANEXO A-4 IMPLEMENTAÇÃO DO DIRETÓRIO

Este anexo apresenta o código parcial que implementa o cluster do Diretório. Também é apresentado o código que implementa a classe de objetos procuradores para o Diretório. Verifica-se a semelhança da implementação do Diretório com um objeto comum comparando com a implementação do cluster para a classe *Cálculo*.

A-4.1 O Cluster Diretório

O cluster do Diretório provê apenas um serviço, prestado pelo método *CriaObjeto*. Neste método há o controle da **identificação das classes** para a criação de objetos. Estes identificadores são *DISTRIBUICAO, CALCULO* e *SAIDA*.

```
/*----------------------- classe DIRETORIO distribuido ----------------*/
class DIRETORIO {

 char comando[MSG_LEN];

 public:
        CtrlMsg COM;

SOCK_ADDR objP;

DIRETORIO             ( void );
~DIRETORIO            ( void );
SOCK_ADDR  CriaObjeto ( int, char * );
};


DIRETORIO::DIRETORIO ( void )
{
 return;
}


DIRETORIO::~DIRETORIO ( void )
{
 return;
}


SOCK_ADDR DIRETORIO::CriaObjeto ( int obj, char * parm ,int tamanho)
{
```

```
   SOCK_ADDR newobj;
   char * endereco;

   switch(obj)
   {
      case DISTRIBUICAO: sprintf(comando,"rsh %s c++/MandelD/temp/distribuicao",
      seleciona_maquina());
      break;

      case CALCULO:      sprintf(comando,"rsh %s c++/MandelD/temp/calculo",
      seleciona_maquina());
      break;

      case SAIDA:        sprintf(comando,"rsh %s c++/MandelD/temp/saida",
         seleciona_maquina());
     break;
   }

   system(comando);

   endereco = COM.RecebeDad(); /* recebe mensagem c/ endereco
do objeto criado */

   newobj = Decode(endereco);    /* decodifica mensagem com endereco
*/

   if (tamanho != 0 ) COM.Envia(parm,tamanho,newobj.addr); /*
envia parametros */
}

/*****************************************************************************/
/*********************Programa que contem o objeto DIRETORIO******************/
/*****************************************************************************/

void main ( )
{
   DIRETORIO * dir;
   char resposta[MSG_LEN];
   CtrlMsg ServMsg;
   MSG * mensagem;


   dir = new DIRETORIO();

   while ( 1 )          /* Delegacao */
   {
      mensagem = ServMsg.RecebeDad();

      switch ( mensagem->metodo )
      {

 case M_DIRETORIO_1 : delete(dir);
      exit(0);

 case M_DIRETORIO_2 : int * tam;
      SOCK_ADDR endereco;

                           tam = (int *)mensagem->parm;

      endereco = dir->CriaObjeto(mensagem->objeto,
                                 &(mensagem->parm[sizeof(int)]),
*tam);
                           sprintf(resposta,"%d %s",endereco.porta,endereco.maquina);
      ServMsg.RespondeDad(resposta,strlen(resposta));
      break;
      }
   }

}
```

A-4.2 Classe Procuradora do Diretório

A classe para objetos procuradores do Diretório é apresentada abaixo.

```
/*---------------- classe procuradora de DIRETORIO ----------------*/
class DIRETORIO_P {

 char parametros[MSG_LEN];
 CtrlMsg COM;

 public:

    SOCK_ADDR objD;

    DIRETORIO_P               ( void );
    DIRETORIO_P               ( int, char * );
    ~DIRETORIO_P              ( void );
    SOCK_ADDR  CriaObjeto     ( int, char*, int );
};

/*-------- definicao da classe procuradora de DIRETORIO ---------*/
DIRETORIO_P::DIRETORIO_P ( void )
{
   char * endereco;

   sprintf(parametros,"rsh %s /home/minuano/rrsantos/c++/MandelD/diretorio",
   "mate");

   system(parametros);

}

DIRETORIO_P::DIRETORIO_P ( int porta, char * maquina )
{
  objD = setaddr(porta,maquina);
}

DIRETORIO_P::~DIRETORIO_P ( void )
{
  int * metodo=1;

  sprintf(parametros,"%d",metodo);
  COM.EnviaAss(parametros,strlen(parametros),objD.addr);
}

SOCK_ADDR DIRETORIO_P::CriaObjeto(int obj, char *parm, int tamanho)
{
   char * endereco;
   int * metodo=2;
   SOCK_ADDR newobj;

   sprintf(parametros,"%d %d %d %s",metodo,obj,tamanho,parm);

   endereco = COM.EnviaSin(parametros,strlen(parametros),objD.addr);

   newobj = Decode(endereco);

   return(newobj);
}
```

BIBLIOGRAFIA

[AGH86] AGHA, G. **Actors**: Concurrent Programming in Distributed Systems. Cambridge: MIT Press. 1987. 144p.

[AHU86] AHUJA, G. Linda and Friends. **Computer**, Los Alamitos, v. 19, n. 8, p. 26–34, Aug. 1986.

[AME87] AMERICA, P. POOL–T A Parallel Object–Oriented Language. In: **Object–Oriented Concurrent Programming**. Cambridge: MIT Press. 1987. p. 197–220.

[ASS92] ASSUMPÇÃO JÚNIOR, J. M. Supercomputador Orientado a Objetos. In: SBAC–PAD, 4., 1992, São Paulo. **Anais**... São Paulo: EPUSP, 1992. p. 337–345,

[AUG92] AUGUSTIN, I. **Paralelismo em Linguagens Orientadas a Objetos**. Porto Alegre: CPGCC–UFRGS, 1992. 86p. (Trabalho individual).

[BAL89] BAL, H. E.; STEINER, J. G.; TANENBAUM, A. S. Programming Languages for Distributed Computing Systems. **ACM Computing Surveys**, New York, v. 21, n. 3, p. 261–320, Sept. 1989.

[BEN87] BENNETT, J. K. The Design and Implementation of Distributed Smalltalk. **SIGPLAN Notices**, New York, v. 22, n. 12, p. 318–330, 1987. (OOSPLA'87).

[BER88] BERSHAD, B. N.; LAZOWSKA, E. D.; LEVY, H. M. PRESTO: A system for Object–Oriented Parallel Programming. **Software Pratice and Experience**, New York, v. 18, n. 8, p. 713–732, Aug. 1988.

[BOR90] BORLAND INTERNATIONAL. **Turbo C++**: Users's Guide. Scotts Valley: Borland International, 1990. 264p.

[BRO86] BRONNENBERG, W. J. H. H. et al. The Architeture of DOOM. In: TRELEAVEN, P.; VANNESCHI, M. (Eds.). **Future Parallel Computers**. Berlin: Spring–Verlag, 1987. 492p. (Lecture Notes in Computer Science, 272).

[CAV92] CAVALHEIRO, G. G. H. **Implementação de Concorrência em C++**. Porto Alegre: CPGCC–UFRGS, 1992. 76p. (Trabalho individual).

[CAV93a] CAVALHEIRO, G. G. H.; NAVAUX, P. DPC++: Uma Linguagem para Processamento Distribuído. In: SBAC-PAD, 5., 1993, Florianópolis. **Anais**... Florianópolis: [s. n.], 1993. p. 732–744.

[CAV93b] CAVALHEIRO, G. G. H.; NAVAUX, P. Um Modelo Distribuído para Linguagens Orientadas a Objetos. In: SEMISH, 20., 1993, Florianópolis. **Anais**... Florianópolis: [s. n.], 1993. p. 518–532.

[CAV93c] CAVALHEIRO, G. G. H.; SANTOS, R. R.; NAVAUX, P. Análise de Desempenho de um Protótipo da Linguagem DPC++. In: SEMISH, 21., 1994, Caxambú. **Anais**... Belo Horizonte: UFMG, 1994. p. 333–347.

[CHI91] CHIN, R. S.; CHANSON, S. T. Distributed Object–Based Programming Systems. **ACM Computing Systems**, New York, v. 23, n. 1, p. 91–124, Mar. 1991.

[ELL90] ELLIS, A. M.; STROUSTRUP, B. **Annotated C++ Reference Manual**. Reading: Addison Wesley, 1990. 447p.

[FLI89] FLIELLER, S. T.Node, Industrial Version of Supernode. **Computer Physics Communications**, Amsterdam, v. 57, p. 492–494, 1989.

[GRE86] GREGORY, S.; CLARK, K. L. Parlog: Parallel Programming in Logic. **ACM Trans. Programming Language Systems**, New York, v. 8, n. 1, p. 1–49, Jan. 1986.

[HOP89] HOPKINGS, T. P.; WOLEZKO, M. I. Writing Concurrent Object–Oriented Programs Using Smaltalk-80. **The Computer Journal**, Cambridge, v. 32 n. 4, p. 341–350, Aug. 1989.

[HUF89] HUFNAGEL, S. P.; BROWNE, J. C. Performance Properties of Vertically Partioned Object–Oriented Systems. **IEEE Transaction on Software Enginnering**, New York, v. 32, n. 4, p. 935–946, Aug. 1989.

[HUN86] HUNDAK, P. Exploring Parafuncional Programming: Separating the What from the How. **IEEE Software**, Los Alamitos, v. 5, n. 1, p. 54–61, Jan. 1986.

[HWA84] HWANG, K.; BRIGGS, F. **Computer Architecture and Parallel Processing**. New York: McGraw-Hill, 1984. 846p.

[KIR91] KIRNER, C. Arquitetura de Sistemas Avançados de Computação. In: JORNADA DE PROCESSAMENTO DE ALTO DESEMPENHO, 1991, São Paulo. **Anais**... São Paulo: EDUSP, 1991. 369p.

[KER78] KERNIGHAN, B. W.; RITCHIE, D. M. **The C Programming Language**. Englewood Cliffs: Prentice–Hall, 1978. 228p.

[LOU92] LOURES, E. F.; SILVA, G. P.; ALMEIDA, V. Análise de Desempenho de Programas Paralelos em Redes de Workstations. In: SBAC–PAD, 4., 1992, São Paulo. **Anais**... São Paulo: EPUSP, 1992. p. 365–377.

[INM84] INMOS LTD. **Occam Programming Manual**. Englewood Cliffs: Prentice-Hall, INMOS LTD, 1984.

[MAR91] MARCHIORO, G. **Comunicação entre Processo no Sistema Operacional Unix**. Porto Alegre: CPGCC da UFRGS, 1991. 31p. (Trabalho individual).

[MEY987] MEYER, B. Eiffel: Programming for Reusability and Extendibility. **SIGPLAN Notices**, New York, v. 26, n. 2, p 85–94, Feb. 1987.

[MUL90] MULLENDER,S.J. et al. Amoeba – A Distributed Operation System for the 1990s. **Computers**, Los Alamitos, v. 23, n. 5, p 44–53, May 1990.

[NET92] NETO, D. et al. Programando em "Linda": Transformações de Espectro. In: SBAC–PAD, 4., 1992, São Paulo. **Anais**... São Paulo: EPUSP, 1992. p. 549–562.

[NIE87] NIERSTRASZ, O. M. Actives Objects in Hybrid. **SIGPLAN Notices**, New York, v. 22, n. 10, p. 7–18, Oct. 1987.

[NYQ92] NYQUIST, E.; HENRICSON, N. **Programming in C++, Rules and Recommendations**. Älvsjö, Seden: ELLEMTEL Telecommunication Systems Laboratories, 1992. (Technical report).

[OKA94] OKAMURA, H.; ISHIKAWA, Y. Object Location Control Using Meta–Level Programming. In: TOKORO, M.; PARESCHI, R. (Eds.). **Object–Oriented Programming**. Berlin: Spring–Verlag, 1994. 540p. (Lecture Notes in Computer Science, 821).

[OSO89] OSÓRIO, F. **Um Estudo Sobre Fractais**: Geração e Usos. Porto Alegre: CPGCC da UFRGS, 1989. 37p. (Trabalho individual).

[ROO86] ROOME, W. D.; GENANI, N. H. Concurrent C. **Software Pratice and Experience**, New York, v. 16, n. 9, p. 821–844, Sept. 1986.

[ROS91] ROSA, F. **Programação Distribuída Baseada em Objetos**. Porto Alegre: CPGCC da UFRGS, 1991. 65p. (Trabalho individual).

[SHA87] SHAPIRO, E. **Concurrent Prolog**: Collected Papers. Cambrige: MIT Press, 1987. 2v.

[SNY86] SNYDER, A. Encapsulation and Inheritance in Object–Oriented Programming Languages. **SIGPLAN Notices**, New York, v. 21, n. 11, p. 38–45, Nov. 1986.

[SNY93] SNYDER, A. The Essence of Objects: Concepts and Terms. **IEEE Software**, Los Alamitos, v. 23, n. 1, p. 31–42, Jan. 1993.

[STR86] STROUSTRUP, B. An Overview of C++. **SIGPLAN Notices**, New York, v. 21, n. 10, p. 7–18, Oct. 1986.

[SUN89] SUN MICROSYSTEMS. **Sun C++ Programer's Guide**. Montain View: SUN, 1989. 284 p.

[SUN90a] SUN MICROSYSTEMS. **Network Programming Guide**. Montain View: SUN, 1990. 353 p.

[SUN90b] SUN MICROSYSTEMS. **SunOs Reference Manual**. Montain View: SUN, 1990. 3v.

[TAK90] TAKAHASHI, T.; LIESENBERG, K. E. Programação Orientada a Objetos. In: **ESCOLA DE COMPUTAÇÃO**, 7., 1990, São Paulo. **Anais**... São Paulo: IME–USP, 1990. 148p.

[YAU92] YAU, S. S.; JIA, X.; BAE, D.-H. Software Desing Methods for Distributed Computing Systems. **Computer Communications**, Oxford, v. 15, n. 4, p. 213–224, May 1992.

[YIN90] YIN, M. -L.; BIC, LUBOMIR; UNGERER, T. Parallelizing Static C++ Programs. In: TOOLS, 3., 1990, Sidney. **Proceedings**..., Sidney: Tools Pacific, 1990. p. 267–275.

[YOK86] YOKOTE, Y.; TOKORO, M. The Design and Implementation of ConcurrentSmalltalk. **SIGPLAN Notices**, New York, v. 21, n. 11, p. 331–340, Nov. 1986. (OOSPLA'86).

[YON86] YONEZAWA, A.; BRIOT, J.-P.; SHIBAYAMA, E. Object–Oriented Concurrent Programming in ABCL/1. **SIGPLAN Notices**, New York, v. 21, n. 11, p. 258–268, Aug. 1986. (OOSPLA'86).

[VAI90] VAIDYANATHAN, P.;MIDKIFF, S. Performance Evaluation of Communication Protocols for Distributed Processing. **Computer Communications**, Oxford, v. 13, n. 5, p. 275–282, June 1990.

[WEG87] WEGNER, P. Dimensions of Object–Based Language Design. **SIGPLAN Notices**, New York, v. 22, n. 12, p. 168–182, Dec. 1987. (OOSPLA'87).

[WEG92] WEGNER, P. Dimensions of Object–Oriented Modeling. **Computer**, Los Alamitos, v. 25, n. 10, p. 12–20, Oct. 1992.

[WYA92] WYATT, B.; KAVI, K.; HUFNAGEL, S. Paralellism in Objetc–Oriented Languages: A Survey. **IEEE Software**, Los Alamitos, v. 9, n. 6, p. 56–66, Nov. 1992.

Informática
UFRGS

Um Modelo para Linguagens Orientadas a Objetos Distribuído.

Dissertação apresentada aos Senhores:

Profa. Dra. Ana Maria de Alencar Price

Prof. Dr. Carlos Alberto Heuser

Prof. Dr. José Lucas Rangel Netto (PUC/RJ)

Vista e permitida a impressão.
Porto Alegre, 21/03/95.

Prof. Dr. Philippe Olivier Alexandre Navaux,
Orientador.

Prof. Dr. Cláudio Fernando Resin Geyer,
Co-orientador.

Prof. Dr. José Palazzo Moreira de Oliveira,
Coordenador do Curso de Pós-Graduação
em Ciência da Computação.