UM ESTUDO DE AMBIENTES DE PROGRAMAÇÃO DISTRIBUIDA: PROPOSTA DE EXTENSÕES PARA MODULA-2

Lidia Micaela Segre

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIAS (D.Sc.) EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof. Michael A. Stanton

Profa. Sueli B. Teixeira Mendes

Prof. Carlos José Pereira de Lucena

Prof. Claudio Kirner

Prof. Valmir Carneiro Barbosa

RIO DE JANEIRO, RJ - BRASIL
DEZEMBRO DE 1987

SEGRE, LIDIA MICAELA

Um estudo de Ambientes de Programação Distribuida: Proposta de extensões para Modula-2 [Rio de Janeiro] 1987.

XVIII, 338 p., 29,7cm (COPPE/UFRJ, D.Sc., Engenharia de Sistemas e Computação (1987).

Tese - Universidade Federal do Rio de Janeiro, COPPE.

1. Sistemas Distribuidos I. COPPE/UFRJ
II. Título (série).

Esta tese é dedicada
- a Rodrigo, Luciano
e Gabriel

AGRADECIMENTOS

Ao Professor Michael A. Stanton, pela orientação, incentivo e apoio durante o desenvolvimento deste trabalho.

À Professora Sueli Mendes, pela co-orientação, mas mais especialmente pelo seu incentivo e apoio emocional.

Ao Professor Claudio Kirner, pelo caminho andado juntos no início de nossas teses.

À Luis Fernando de Melo Campos, Esther Frankel e Vania Didier que me ajudaram a crescer como pessoa nestes últimos anos.

À Susana Scheimberg de Makler, Graciela Galizia de Tagliaferri, Luiz Julião Braga Filho, Ana Lucia Cavalcanti Wanderley, Alberto Noé e Miriam Marques que me acompanharam durante a longa fase de desenvolvimento deste trabalho apoiando-me, incentivando-me e confiando em mim.

À muitos outros amigos, que mesmo não nomeando-os individualmente, não quer dizer que tenham sido esquecidos.

Aos meus filhos pela paciência e compreensão que tiveram, e às minhas empregadas e amigas, Maria do Nascimento e Maria Lucia Borges Pinto, que muitas vezes me substituiram nas minhas obrigações de mãe e dona de casa, além do carinho e apoio que me brindaram.

Ao Programa de Engenharia de Sistemas e Computação pela liberação concedida, e aos meus colegas de trabalho que me incentivaram e me substituiram em algumas tarefas durante o período mais intenso deste trabalho.

Aos alunos que trabalharam junto comigo, Vanise Vetromille, Nanci Lages e Nelson da Silva.

À UFRJ, FINEP, Embratel e PUC/RJ que apoiaram este trabalho.

À Ruth Maria de Barros Fagundes de Sousa pela paciência e dedicação na datilografia.

Resumo de Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para obtenção do grau de Doutor de Ciências (D.Sc.)

UM ESTUDO DE AMBIENTES DE PROGRAMAÇÃO DISTRIBUIDA: PROPOSTA DE EXTENSÕES PARA MODULA-2

Lidia Micaela Segre

Dezembro, 1987

Orientador: Michael Anthony Stanton
Programa: Engenharia de Sistemas e Computação

Este trabalho descreve a definição e implementação de um ambiente de programação distribuida, baseado numa linguagem de programação amplamente divulgada e aceita. Para isto foram identificadas as características mais importantes dos sistemas distribuidos, grandes e modulares, e foi possível reconhecer os dois níveis habituais de programação: programação em pequena escala (utilizando uma linguagem chamada de LPP), dos componentes básicos do sistema, e programação em larga escala onde é definida a configuração do sistema, através da especificação dos componentes que o constituem e de suas ligações. Esta definição pode ser feita de maneira estática ou dinâmica, através de uma linguagem separada (chamada de LPL) ou não.

Foi escolhida como linguagem base Modula-2 e, depois de um estudo detalhado das características das LPP, decidiu-se acrescentar, como mecanismo de comunicação e sincronização entre processos, chamada remota de procedimento (RPC) de forma transparente ao usuário.

São analisadas as propriedades necessárias para a fase de

configuração, e são levantados os problemas que precisam ser tratados tanto na configuração estática como dinâmica. A fim de completar o ambiente de programação distribuida, foi definida uma linguagem separada de configuração estática, a ser utilizada com a linguagem Modula-2 estendida. Sua definição e implementação foram feitas tendo em vista uma futura adaptação para configuração dinâmica. A comparação desta proposta com várias outras encontradas na literatura permite por um lado mostrar o seu potencial, e por outro indicar os passos futuros a serem realizados para obter um ambiente de programação distribuida robusto e confiável.

Abstract of Thesis presented to COPPE/UFRJ as partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

A STUDY OF DISTRIBUTED PROGRAMMING ENVIRONMENTS: EXTENSION PROPOSAL FOR MODULA-2

Lidia Micaela Segre

December, 1987

Adviser: Michael Anthony Stanton
Department: Engenharia de Sistemas e Computação

This work describes the definition and implementation of a distributed programming environment, based on a well-known and accepted programming language. With this end in mind, the main features of large, modular, distributed systems are identified, and, the two usual levels of programming were recognized: programming-in-the-small, using a language (LPS) to program basic components of the system, and programming-in-the-large, where system configuration is defined, possibly through a separate language (LPL), specifying its components and the links among them. This specification can be done either dynamically or statically.

Modula-2 was chosen as the base language, and after a detailed study of LPS features of several languages, it was decided to add to it a transparent remote procedure call (RPC) mechanism for process communication and synchronization.

An analysis is also made of those properties needed for configuring a system, identifying the problems that need to be tackled for static as well as for dynamic configuration.

In order to complete the distributed programming environment, a configuration language is defined, to be used together with extended Modula-2. Its definition and implementation were performed whilst bearing in mind future extensions to cope with dynamic configuration.

An extensive comparison was also made between the present work and others described in the literature, with the aim of showing its potential as well as suggesting further steps towards the building of robust and reliable distributed programming environment.

ÍNDICE

ÍNDICE DE FIGURAS

CAPÍTULO VI

CAPÍTULO VII

CAPÍTULO I

INTRODUÇÃO

A revolução em microeletrônica, que continua reduzindo enormemente o custo de componentes de computadores, já transformou para sempre a prática de computação para a maioria dos usuários. O ambiente característico de computação hoje é baseado no computador pessoal, que oferece uma série de vantagens importantes quando comparado com seus antecessores, os terminais "burros" ligados a grandes sistemas centralizados de partilha de tempo. Estas vantagens, todas decorrentes da presença de "inteligência local", incluem um desempenho constante e portanto previsível, o uso de interfaces mais simples como "menus" ou "janelas", a opção cômoda do uso de disquetes como meio de armazenamento pessoal de informação, altamente transportável entre computadores pessoais diferentes, e uma quantidade de software de apoio nunca antes visto. Embora sejam abundantes os recursos disponíveis dentro dos limites de um computador pessoal, estes podem ser complementados através da ligação destas máquinas entre si a outras mais potentes em redes, dando acesso a serviços e informação disponíveis remotamente.

Uma transformação parecida está sendo observada na área de aplicação de computadores ao controle de processos na indústria, nos transportes e na área de saúde, entre outras, onde a fácil disponibilidade de componentes sofisticados e baratos permite a construção de controladores e sensores bem inteligentes, ligados em rede a unidades de comando e controle.

Os usuários destes sistemas de computação dispõem de conjuntos empacotados de utilitários de redes, tais como acesso remoto a terminais, transferência de arquivos, correio eletrônico e algumas aplicações distribuídas de base de dados. A tarefa de escrever o software para usar as redes é geralmente deixada para os programadores especialistas de sistemas.

Mais recentemente, com a disponibilidade de estações de trabalho e redes locais com preços cada dia menores, surgiu um forte incentivo para que os usuários desenvolvam seus próprios programas distribuídos. Assim como os sistemas distribuídos são

cada dia mais comuns, as aplicações distribuidas também. A programação distribuida não é mais uma área reservada para um pequeno grupo de programadores especialistas.

Estas novas áreas de aplicação criam desafios para a construção de software, e particularmente para o processo de programação. Em particular, são apresentados novos requisitos, ou da linguagem de programação utilizada, ou das ferramentas a serem empregadas no desenvolvimento do software, ou do ambiente de execução a ser usado.

Enquanto as vantagens do uso de sistemas distribuidos são bem conhecidas e fortemente aclamadas, não existe ainda um consenso sobre como prover o suporte necessário para modularidade, concorrência, sincronização, comunicação e configuração. As soluções propostas variam desde as tentativas de simplesmente adaptar e interligar sistemas autônomos já existentes, até as propostas de fornecer um ambiente completo baseado numa linguagem na qual possam ser escritos os programas distribuidos.

Podemos encontrar vários sistemas operacionais de redes, como é mostrado por TANENBAUM em [163], que provêem acesso às ferramentas de comunicação. Neste caso uma aplicação distribuida é implementada por um conjunto de programas sequenciais que se comunicam usando as chamadas de sistemas da rede. Como foi apresentado por MAGEE et alii em [106], estes sistemas apresentam uma série de desvantagens tais como:

i) as interfaces de comunicação são complexas e difíceis de usar.

ii) as convenções de nomeação e as primitivas de comunicação entre processos não são geralmente uniformes.

iii) é provido pouco suporte para a configuração inicial de um conjunto de componentes de um programa numa aplicação distribuida e para monitoramento e controle posterior da configuração.

iv) pouca ou nenhuma verificação das interfaces é provida para garantir compatibilidade dos programas interconectados.

As principais vantagens destes sistemas operacionais de

redes são por um lado a sua flexibilidade, já que um sistema distribuido é composto de um conjunto, potencialmente modificável, de programas interligados, e, por outro, o fato de estar baseado em sistemas operacionais já estabelecidos, oferecendo uma grande variedade de serviços e utilitários que estão já acessíveis e utilizáveis.

Num outro enfoque, as linguagens de programação distribuida, tais como ARGUS (LISKOV [99]), CONIC (KRAMER, MAGEE e SLOMAN [106, 87, 105, 152]), SR (ANDREWS [4]), "Distributed Processes" (BRINCH HANSEN [29]), *MOD (COOK [42]), Mascot3 (BATE [16], SIMPSON [151]), reduzem a complexidade da construção de aplicações distribuidas provendo ferramentas para modularidade, concorrência, sincronização e comunicação, integradas na mesma linguagem. Elas provêem suporte para verificação na compilação, ligação, e em tempo de execução de maneira a assegurar a compatibilidade de mensagens ou operações entre componentes. Elas provêem também consistência na nomeação, comunicação e sincronização tanto para interações locais quanto remotas. Portanto os ambientes das linguagens são geralmente mais simples e mais confiáveis. Entretanto os acessos diretos às ferramentas do sistema residente são geralmente difíceis. Na maioria dos casos as características de configuração fazem parte da linguagem de programação, produzindo um único programa grande distribuido, em lugar do enfoque anterior de um sistema como um conjunto de programas interligados, sujeitos a mudanças.

A partir destas idéias foram desenvolvidos ambientes de programação distribuidos baseados em linguagens de programação, que foram estendidos com as características vantajosas do enfoque de sistemas operacionais de rede, para serem acrescentadas às já mencionadas dos ambientes de linguagens. Podemos citar como exemplo o sistema Cnet [40] baseado na linguagem ADA, e o sistema CONIC [106], no qual foi definida uma linguagem de programação a partir de Pascal e uma linguagem de configuração separada. Estes exemplos, mesmo tendo filosofias diferentes, que serão analisadas com detalhe nos próximos capítulos, nortearam o nosso trabalho.

O nosso objetivo foi o de desenvolver um ambiente de programação distribuida baseado numa linguagem já reconhecida internacionalmente, que possuisse uma série mínima de características adequadas, a fim de poder acrescentar as

extensões necessárias para construir o ambiente desejado. Uma característica muito ponderada na escolha da linguagem foi a sua disponibilidade e facilidade de implementação em computadores pequenos e nacionais. Passaremos a apresentar em seguida os outros argumentos que determinaram a escolha da linguagem.

Os principais aspectos da programação registraram progressos na década dos 70. Podemos citar a evolução do conceito de tipo de dado a partir da contribuição de Pascal (WIRTH [175]) até os tipos abstratos; a formalização e implementação de modelos de programação concorrente; a formalização dos conceitos de modularidade e compilação separada; a percepção da importância da transportabilidade de software e a tecnologia de implementação de compiladores que a facilita; e o reconhecimento de que o processo de programação requer o apoio de um ambiente de suporte que auxilia todas as atividades de programação do ciclo de vida de um software.

O final da década viu um grande trabalho de síntese no qual foram propostas soluções para o problema de programação que reuniam vários dos diversos aspectos supracitados. Entre as várias linguagens que apareceram e que faziam uma espécie de síntese dos avanços em linguagens de programação dos dez anos anteriores podemos destacar particularmente duas que foram as mais aceitas e divulgadas internacionalmente: ADA e Modula-2.

ADA (ICHBIAH [73]) pode ser caracterizada de forma sumária como Pascal com extensões para atender à modularização (pacotes), á programação concorrente, ao tratamento de exceções e à parametrização de módulos por tipo de dados (unidades genéricas). ADA foi criada para ser uma linguagem para a programação de sistemas dedicados ("embedded systems") para aplicações militares do governo norteamericano. Modula-2 (WIRTH [178, 180]) acrescenta a Pascal os conceitos de modularização e co-rotina. Esta linguagem é uma evolução direta de Pascal, através de Modula (WIRTH [176]), que foi a primeira tentativa de WIRTH de criar uma linguagem para programar pequenos sistemas dedicados. Tanto ADA como Modula-2 são destinadas ao desenvolvimento profissional de software, e, como acabamos de ver, inicialmente destinadas para a programação de sistemas dedicados. Porém, ambas estas linguagens, além de possuirem as características específicas para este ambiente, também possuem toda a funcionalidade necessária para a

programação de aplicações mais convencionais. Aliás, com o suporte dado para desenvolvimento modular, ambas estas linguagens são altamente apropriadas para praticamente qualquer área de aplicação, e certamente para qualquer uma hoje programada em C (KERNIGHAN e RITCHIE [79]) ou Pascal.

As facilidades mais substanciais de ADA que estão ausentes em Modula-2 são a definição de operadores e unidades genéricas, que sem dúvida contribuem ao poder de abstração da linguagem, mas exige um preço elevado em termos da complexidade da implementação. Por outro lado, Modula-2 provê facilidades ausentes em ADA para o manuseio direto dos mecanismos de interrupção e de entrada/saída, permitindo escrever o software mais fundamental nesta mesma linguagem. Estas facilidades permitem implementar diferentes políticas no nível do núcleo, como por exemplo, para gerenciamento do processador e dos processos, ou em outros níveis para gerenciamento de memória e de periféricos. Em contraposição ADA já inclui políticas de gerenciamento embutidas que não podem ser modificadas pelo usuário. Esta vantagem de Modula-2 rende a linguagem mais flexível permitindo ao usuário escolher as políticas mais convenientes para cada aplicação. A conclusão é que Modula-2 é uma linguagem que compete com ADA nas aplicações para as quais ADA foi projetada. Além disto, Modula-2 foi projetada por uma só pessoa, ao invés de uma comissão, e com grande sensibilidade para os problemas de implementação.

O processo de criação de ADA também deu uma contribuição significativa à conceituação de ambientes de suporte de programação. No relatório Stoneman (DOD [50]) foram especificadas de forma clara e concisa as características que deveria ter o software de suporte ao processo de desenvolvimento de programas em ADA. Embora Stoneman tenha sido escrito tendo ADA em mente as idéias apresentadas são imediatamente aplicáveis a qualquer linguagem que dê suporte a modularização, por exemplo, Modula-2.

Finalmente, se desejarmos usar uma determinada linguagem de programação em ambientes distribuídos, será necessário definir soluções para uma série de problemas pertinentes, dos quais os mais sérios dizem respeito à comunicação entre processos em computadores diferentes, ao tratamento de falhas, e à carga e ligação de software distribuído. Para resolver estes problemas

devem ser definidos mecanismos, ao nível da própria linguagem, ou na forma de extensões a esta, ou possivelmente na forma de uma notação ou linguagem de nível mais alto que a de programação.

Acreditamos que Modula-2, por sua simplicidade e elegância, é uma ferramenta eficaz para a produção de software industrial nos dias de hoje. Adicionalmente, é superior a Pascal para fins didáticos, como é evidenciado pela sua adoção para ensino básico em universidades como Imperial College, Londres, Universidade de Genebra, Suiça, entre outras.

Uma vez escolhida a linguagem Modula-2 o projeto de construir o ambiente de programação distribuido foi desenvolvido por um grupo de professores e alunos da PUC/RJ e da UFRJ, produzindo uma série de artigos apresentados em congressos nacionais e internacionais [144, 143, 156, 157, 145, 146] e várias teses de mestrado [45, 88, 170] e esta tese de doutorado.

O projeto, de forma resumida, consiste de três partes:

i) a implementação de Modula-2 e seu suporte de programação em micro de 8 bits [135];

ii) a definição e implementação de mecanismos de comunicação e sincronização para nós frouxamente acoplados num sistema distribuido, que foram acrescentados à linguagem [145, 45, 88];

iii) a definição e implementação de uma linguagem de configuração estática necessária para definir os componentes do programa distribuido, estabelecer as suas ligações e fazer o carregamento na arquitetura física na qual será executado [146].

Esta tese apresenta particularmente os dois últimos pontos, fazendo uma discussão exaustiva das motivações e das justificativas que levaram à escolha das definições e implementações do mecanismo de comunicação e sincronização entre processos denominado chamada remota de procedimento e da linguagem separada de configuração estática a ser acrescentada à Modula-2.

Este trabalho foi organizado em vários capítulos que descrevemos resumidamente. No Capítulo II foi feita uma análise

dos diferentes tipos de sistemas distribuídos que foram desenvolvidos nos últimos anos procurando apresentar algumas características principais. É discutida também a importância da estruturação modular de programas distribuídos e apresentado o conjunto de requisitos necessários para uma linguagem de programação distribuída.

O Capítulo III se concentra nas características de programação em pequena escala, analisando os diferentes modelos utilizados para programar os módulos que compõem o sistema distribuído, e em particular os mecanismos de comunicação e sincronização entre processos localizados em nós físicos distintos. Por último é discutida a dualidade dos dois modelos nos quais pode ser caracterizada a maioria dos mecanismos encontrados na literatura.

O objetivo do Capítulo IV é apresentar a nossa proposta de programação em pequena escala. Para isto é primeiro apresentada uma análise de outras propostas existentes e as nossas justificativas para a escolha tomada. São apresentados alguns núcleos de multiprogramação implementados usando Modula-2. A seguir é feito um estudo detalhado do mecanismo de chamada remota de procedimento para depois descrever a nossa implementação.

No Capítulo V são analisadas as características principais do conceito de configuração a partir das linguagens de interconexão modular utilizadas nos projetos de engenharia de software. É realizada uma análise sobre o modelo de configuração estática e configuração dinâmica para a seguir fazer um levantamento sobre as propriedades necessárias para configuração. Estas propriedades estão relacionadas com a linguagem de programação, a linguagem de configuração e a especificação das mudanças, o processo de validação, o gerenciamento de configuração e a unidade de configuração e substituição. Finalmente é feito um estudo sobre as condições necessárias para reconfiguração.

No Capítulo VI é apresentada a definição da linguagem de configuração estática proposta para ser acrescentada a Modula-2, a fim de obter o ambiente para programação distribuída. Esta apresentação é ilustrada através de uma série de exemplos que mostram a potencialidade da linguagem. A seguir é descrita a implementação do interpretador da linguagem de configuração, e as

suas interfaces com a implementação do mecanismo de comunicação e sincronização para chamadas remotas de procedimentos.

O Capítulo VII tem como objetivo comparar o nosso modelo com outras propostas encontradas na literatura.

Para finalizar são discutidos alguns conceitos em relação aos problemas levantados, particularmente para a implementação de configuração dinâmica, área que ainda está mais em aberto. São apontados também os passos futuros que deveriam ser abordados para a obtenção do ambiente de programação distribuída desejado.

CAPÍTULO II

SISTEMAS DISTRIBUIDOS

A contínua e rápida queda dos custos de equipamentos de computação possibilita a organização de sistemas de computação na forma dita "Sistemas Distribuidos". Com isto surgem possibilidades de exploração de novas organizações de sistemas computacionais e também de novos problemas abrindo um vasto campo para pesquisa, como foi apontado por SALTZER em [138].

Existe um consenso quanto à importância cada vez maior que terão no futuro os sistemas distribuidos. Infelizmente não existe o mesmo consenso em relação ao significado do termo "Sistema Distribuido".

Por exemplo, existem diferentes maneiras de ligar máquinas iguais ou diferentes para formar, desde um conjunto de estações de trabalho ligadas entre si através de uma rede de comunicação, até a ligação de máquinas com o objetivo de obter uma máquina única mais poderosa, sendo transparente ao usuário a sua arquitetura e a sua maneira de compartilhar recursos. Os problemas a serem encarados estão ligados com o tratamento do paralelismo, a comunicação e sincronização de programas rodando em máquinas diferentes, os gerenciadores de recursos compartilhados, a transparência ou não do sistema, a proteção, o tratamento de falhas, etc.

Na California em dezembro de 1980, houve um encontro promovido pela ACM ("Association for Computing Machinery"), sobre questões fundamentais em Computação Distribuida [96]. Durante três dias foram discutidos os seguintes temas: Sistemas Existentes, Atomicidade, Proteção, Aplicações, Nomeação, Comunicações, O que é que precisamos da teoria?, Quais são os problemas práticos importantes? Na última sessão denominada "O que é que é diferente em Computação Distribuida", SHOCK, depois de dar argumentos contra e a favor das diferenças, encerrou a discussão reconhecendo que não tinha uma definição precisa de um sistema distribuido, do que faz um sistema distribuido, do que faz com que ele seja diferente de qualquer outro tipo de sistema, e que nessa hora era melhor apelar a uma autoridade de mais alto nível. Nesse caso ele sugeria o Supremo Tribunal dos Estados

Unidos, que já tinha sido confrontado com um problema similar quando procurou definir "obscenidade ou pornografia", e o magistrado STEWART fez a famosa observação, que segundo SHOCK pode ser aplicada para a definição de um Sistema Distribuído:

"Eu não sei como defini-lo, mas sei reconhecê-lo quando o vejo".

II.1 ALGUMAS DEFINIÇÕES

O conceito de Sistema Distribuído tem sido largamente aplicado, com diversos significados, e parece-nos que o apresentado em [181] por YAN et alii, é o mais abrangente e completo, embora até certo ponto vago ainda: "o termo Sistema Distribuído de Computação representa uma grande classe de sistemas desse tipo, desde uma rede estrela de computação até um sistema completamente descentralizado. Em todos os casos o termo distribuído refere-se ao fato de que, unidades de processamento, funções, dados, controle, ou uma combinação destes, são distribuídos em uma certa medida".

É considerado em [181] que sistemas distribuídos de computação têm as seguintes características:

- o sistema tem mais de um processador
- existem elos de comunicação entre os processadores
- existem componentes funcionais que residem em cada processador, e os componentes de software de diferentes processadores podem ser síncronos ou assíncronos
- existem interações entre componentes funcionais que residem em processadores diferentes com a finalidade de executar funções globais, tais como compartilhamento de recursos (incluindo tanto recursos de hardware como de software) e sincronização de execução.

Para tornar mais clara a discussão é interessante destacar que podemos diferenciar três níveis nos quais identificamos a distribuição de um sistema:

i) no nível do hardware e da arquitetura do sistema, quando o sistema está composto de mais de um processador com

memória e periféricos distribuídos;

ii) no nível do sistema operacional que gerencia a arquitetura distribuída, quando as suas funções estão localizadas em máquinas diferentes;

iii) no nível dos aplicativos, isto é, do software básico desenvolvido por cima do sistema operacional (por ex. compilador), ou nas aplicações específicas (por ex. para controle de processos).

Esta distinção de níveis será referenciada nas diferentes formas encontradas na literatura para classificar sistemas distribuídos que analisaremos a seguir.

Em [76], JONES e SCHWARZ levantam a necessidade de definir o termo "distribuição" que por ter sido tão confusamente usado em relação a software e hardware, seu significado ficou obscuro. Para eles distribuição é um tipo de organização que se aplica tanto no sentido físico quanto lógico.

Eles definem que um sistema é **fisicamente distribuído** se pelo menos um de seus componentes físicos autônomos está duplicado. Esta definição é muito fraca já que pode incluir alguns computadores que não seriam normalmente considerados distribuídos, como por exemplo, um monoprocessador com vários controladores de disco ou um computador monoprocessador com vários terminais. A definição de **autônomo** é crítica e pode variar de uma aplicação para outra. Esta definição se refere unicamente ao primeiro nível que acabamos de distinguir.

Quanto à **distribuição lógica**, consideram que um sistema logicamente distribuído é construído por objetos ativos chamados processos, e objetos passivos contendo dados ou código. Na sua definição um sistema de programação é logicamente distribuído se cada componente é autônomo de forma que a exclusão de um componente não leve o sistema a falhar na execução da sua tarefa de forma inaceitável. Aqui também a definição do aceitável ou inaceitável é importante e pode variar de uma situação para outra. Novamente destacamos que esta definição se aplica aos dois últimos níveis diferenciados anteriormente.

Relatando a discussão do Workshop sobre Computação Distribuída de outubro de 1978 em Cambridge, Massachusetts [127],

PETERSON considera a autonomia dos componentes distribuídos de um sistema distribuído, como requisito fortemente necessário. Por autônomo ele entende a possibilidade de poder remover o componente do sistema sem afetar o funcionamento do resto, e, de forma mais geral, considera que os algoritmos internos e a organização da informação deve poder ser livremente selecionada em cada nó, independentemente dos outros. As decisões em relação a que informação deve ser mantida, como deve ser organizada, como deve ser processada e com que objetivos pode ser usada, devem ser todas tomadas localmente.

PETERSON coloca também que durante as discussões chegou-se à conclusão de que a distribuição lógica é mais importante que a física. A distribuição lógica pode ser implementada num único processador ou em vários processadores num mesmo lugar ou em vários processadores em diferentes lugares; ela é independente da distribuição física.

Para enfatizar o conceito de sistemas logicamente distribuídos, daremos alguns exemplos que mostram claramente a diferença entre este conceito e o de sistemas **logicamente centralizados**. Estes exemplos pertencem ao segundo e terceiro nível de diferenciação, isto é, à área de software.

Um compilador que usa uma arquitetura do tipo "pipeline" é composto por processos separados que executam a análise léxica, a análise sintática, a análise semântica e a geração de código. Estes processos podem ter acesso a dados compartilhados ou podem se comunicar através de mensagens. Em qualquer caso a saída de um processo é a entrada para o próximo. Se os diferentes processos do compilador executam em processadores separados de um multiprocessador ou de uma rede, podemos dizer que o compilador é fisicamente distribuído. Contudo, o compilador não é logicamente distribuído. A remoção de um dos processos causaria o não funcionamento do compilador.

Uma estrutura de dados, dependendo da sua organização, pode ser logicamente distribuída ou não, independentemente de onde residam fisicamente seus componentes. Para tornar isto claro consideremos uma estrutura de dados que está organizada como um vetor de entradas, como por exemplo, uma tabela de descritores de segmentos. Se cada descritor pode ser lido e alterado independentemente dos outros, a estrutura de dados é logicamente

distribuída. Se, por outro lado, a tabela inclui uma variável, que precisa ser consultada e atualizada antes de cada escrita (por exemplo um "lock"), então a estrutura de dados é logicamente centralizada.

II.2 TIPOS DE SISTEMAS DISTRIBUIDOS

Apesar de não se ter uma definição consensual de sistemas distribuídos, como visto no ítem anterior, encontram-se na literatura várias maneiras de caracterizá-los. Podemos classificá-los segundo a forma de interligação entre processadores e memória, em sistemas fortemente acoplados e sistemas frouxamente acoplados.

Sistemas **fortemente acoplados** são compostos por vários processadores interligados através de compartilhamento de memória. Esta arquitetura permite que todos os processadores do sistema tenham acesso às memórias e executem o código nelas armazenado. Nestes sistemas geralmente recursos de entrada e saída e periféricos são compartilhados também por todos os processadores. A latência de comunicação entre processadores é baixa pois o tempo de acesso é limitado somente pelo acesso à memória.

Sistemas **frouxamente acoplados** possuem memórias com espaços de endereçamento disjuntos; isto é, os processadores não compartilham uma memória principal comum. No nível do hardware deve existir então uma interface explícita de comunicação entre os processadores e a latência de comunicação entre eles é mais alta que no caso anterior.

Sistemas fortemente acoplados geralmente precisam de mecanismos de sincronização para executar os processos de forma cooperativa, enquanto que sistemas frouxamente acoplados podem executar os processos concorrentes de forma assíncrona. Esta classificação se refere ao primeiro nível descrito no início.

Outra maneira de diferenciar os sistemas distribuídos é usada em [172], onde WEITZMAN classifica sistemas de acordo com a maneira de fazer compartilhamento: para ele os sistemas ou executam no modo de compartilhamento de carga, ou no modo de compartilhamento de recursos, ou em uma combinação dos dois.

Um sistema que funciona no **modo de compartilhamento de carga** é um sistema composto por um conjunto de computadores similares, cada um executando uma unidade básica do trabalho. Se não todos estes computadores estiverem ocupados na hora de ser criada uma nova unidade de trabalho, ela será alocada a um dos computadores ociosos.

Um sistema que funciona no **modo de compartilhamento de recursos** é um sistema composto por um conjunto de computadores diferentes, cada um sendo funcionalmente especializado e provendo recursos para serem usados pelos outros componentes do sistema. Podem ser compartilhados também periféricos únicos ligados a diferentes computadores do sistema.

Podemos ainda classificar os sistemas distribuídos em relação a seu uso. Definimos como **sistemas dedicados** os sistemas desenvolvidos para uma determinada aplicação, como por exemplo um sistema comercial que controla compras, vendas e estoques de várias lojas de uma mesma firma, ou um sistema bancário de controle de várias agências de um mesmo banco. Estes sistemas são geralmente compostos por um conjunto de computadores iguais ou da mesma família, e são chamados **sistemas homogêneos**. Existem também sistemas dedicados para aplicações de controle de processos, ou sistemas de tempo real controlando um determinado processo industrial ou uma unidade hospitalar, nos quais é preciso utilizar equipamentos diferentes para determinadas funções do processo que está sendo controlado. Estes sistemas são chamados **sistemas heterogêneos** e geralmente são mais complexos que os homogêneos, devido às diferenças tanto no software quanto no hardware dos componentes envolvidos.

Em contraposição aos sistemas dedicados, encontramos hoje cada vez mais **sistemas de uso geral** sendo desenvolvidos pela indústria, pelas instituições educacionais e pelas instituições militares. Estes sistemas são geralmente heterogêneos, para poder explorar ao máximo o particionamento de tarefas, tanto no hardware quanto no software, e explorar as capacidades específicas dos computadores utilizados. Sistemas heterogêneos são criados também para satisfazer às necessidades de compartilhar sistemas de informação, que no passado eram utilizados em aplicações dedicadas, mas que a partir de um determinado momento, precisavam ser interligados, como no caso de

bibliotecas que funcionavam independentemente e agora estão ligadas entre si. Podemos citar uma rede de computadores de nível nacional interligando computadores diferentes espalhados geograficamente ou ligando redes locais entre si para uso geral, racionalizando os recursos disponíveis. É o caso de nós que correspondem a centros autônomos de computação interligados através de uma rede de comunicação à distância, como por exemplo centros de processamento universitários e centros de pesquisa.

Nesta classificação a homogeneidade e a heterogeneidade se referem ao primeiro nível da diferenciação inicial, isto é, à arquitetura do sistema, enquanto a classificação de sistemas dedicados ou de uso geral se refere ao último nível antes mencionado, que é o nível dos aplicativos.

É necessário salientar aqui que os conceitos de homogeneidade e heterogeneidade são utilizados também para classificar os sistemas distribuídos no segundo e terceiro nível, isto é, no nível do software do sistema, com um sentido completamente diferente.

Em geral, nestes níveis um sistema é chamado **homogêneo** quando existe uma transparência em relação à arquitetura subjacente, que será composta de nós físicos distintos. Isto é, tanto para o projetista do sistema operacional como para o projetista de uma aplicação, deve ser transparente a rede de computadores sobre a qual ele vai rodar seu software. Desta forma, para o projetista o sistema se apresenta como uma única máquina poderosa.

Podemos citar alguns exemplos de sistemas operacionais distribuídos homogêneos encontrados na literatura, tais como o "Distrix" apresentado por CHRISTIE, que é uma extensão do sistema UNIX versão V, o "Newcastle Connection" apresentado por MARSHALL [32] e o "Apollo Domain" [92, 120] apresentados no Workshop intitulado "Operating Systems in Computer Networks", na Suiça, em janeiro de 1985 [161].

É interessante analisar, vendo os sistemas distribuídos apresentados no Workshop titulado "Making Distributed Systems Work", promovido pela ACM-SIGOPS em Amsterdam, em setembro de 1986, como a tendência dessa homogeneidade está crescendo cada dia mais. Vários dos sistemas que estão sendo desenvolvidos consideram a possibilidade de integrar nós com sistemas

operacionais diferentes e até com linguagens de programação distintas, através da definição de interfaces apropriadas. Este é o caso por exemplo dos sistemas tais como o "Common System Project" apresentado por BLOOM, que é uma extensão do "Newcastle Connection", o "Apollo Domain", que também foi estendido para uma arquitetura heterogênea, e o "The Saguaro Distributed Operating System", que considera o uso de várias linguagens diferentes [8].

II.3 SISTEMAS OPERACIONAIS DISTRIBUIDOS

Consideramos agora outro ponto de vista em relação ao conceito de sistema distribuido, definido em [163] por TANENBAUM e VAN RENESSE. Os autores se concentram no segundo nível da diferenciação inicialmente colocada, isto é, nos **sistemas operacionais distribuidos**, os quais, apesar de possuir características específicas, têm um conjunto de outras em comum com os sistemas distribuidos já mencionados.

Um sistema operacional distribuido é definido por eles como um sistema, que gerencia os múltiplos recursos distribuidos em nós fisicamente separados, mas que se apresenta aos seus usuários como um sistema centralizado; isto corresponde à definiçao do conceito de homogeneidade apresentado para os últimos dois níveis de diferenciação que acabamos de apresentar. O conceito chave é a **transparência**: o uso de vários processadores deve ser invisível ao usuário. Outra maneira de expressar a mesma idéia é dizer que os usuários vêem o sistema como um "monoprocessador virtual" e não como uma coleção de máquinas distintas. Este conceito será tratado com mais profundidade numa outra seção deste capítulo.

Para eles, é o software e não o hardware que determina se um sistema operacional é distribuido ou não. Como regra geral, se o usuário pode dizer que computador está usando, então o sistema operacional implementado não é distribuido.

Para enfatizar as características de um sistema operacional distribuido, eles o comparam com outro tipo de sistema chamado de **sistema operacional de rede**. Uma configuração típica para um sistema operacional de rede é uma coleção de computadores pessoais com um servidor comum de impressão e outro de arquivos, todos ligados por uma rede local. Este sistema tem as seguintes características que o diferenciam de um sistema distribuido:

- cada computador possui seu próprio sistema operacional, em lugar de rodar uma parte de um sistema global;

- cada usuário trabalha normalmente na sua própria máquina; para usar outra máquina ele precisa emitir um pedido de uso do nó remoto; não é o sistema operacional que automaticamente aloca processos aos processadores;

- os usuários sabem onde estão armazenados seus arquivos e precisam mover os arquivos entre máquinas com comandos explícitos de transferência, de modo que o sistema operacional não gerencia a alocação de arquivos;

- o sistema não tem praticamente tolerância a falhas, isto é, se uma porcentagem, mesmo pequena, de computadores pessoais pararem, essa mesma porcentagem de usuários ficarão fora da rede; não é possível que todos os usuários continuem, mesmo com uma pequena degradação no desempenho de cada um.

O ponto chave que distingue um sistema operacional de rede de um sistema operacional distribuído é o fato do usuário ser ciente de estar usando várias máquinas. A visibilidade ocorre em três áreas principais: sistema de arquivos, proteção e execução de programa. Existem sistemas com um alto grau de transparência numa área mas não em outra; estes são chamados de sistemas híbridos segundo esta caracterização.

Gostaríamos ainda de apresentar a caracterização colocada por TANENBAUM e VAN RENESSE em [163] para os diferentes tipos de sistemas operacionais distribuídos, lembrando que a definição adotada por eles só considera como sistemas operacionais distribuídos aqueles sistemas centralizados que executam em processadores múltiplos e independentes de forma transparente ao usuário. Segundo eles esses sistemas podem ser separados em três categorias, chamadas de modelo de minicomputador, modelo de estação de trabalho e modelo de conjunto de minicomputadores.

No **modelo de minicomputador**, o sistema consiste de alguns minicomputadores (da ordem de uma dúzia) onde cada um atende a vários usuários. A cada usuário é alocada uma determinada máquina e ele tem acesso remoto às outras. Este modelo é uma simples extensão da máquina usada no modo de tempo compartilhado ("time-

sharing").

No **modelo da estação de trabalho**, cada usuário tem uma estação de trabalho pessoal geralmente equipada com um processador poderoso, memória, um console com facilidades gráficas e às vezes um disco. Quase todo o trabalho é executado na própria estação de trabalho. Este sistema pode ser considerado distribuído, se ele der suporte a um único sistema global de arquivos de maneira que se possa ter acesso aos dados independentemente de sua localização.

O **modelo de conjunto de minicomputadores** é o passo seguinte nessa evolução, depois do modelo de estação de trabalho. À medida que as unidades centrais de processamento vão tendo um preço menor, este modelo será cada vez mais difundido. A característica principal é que, havendo vários processadores livres, um ou mais deles são alocados temporariamente ao usuário à medida que ele vai precisando de um poder maior de processamento. Quando o trabalho é completado, os processadores alocados são liberados e devolvidos ao conjunto dos livres para esperar a próxima solicitação.

II.4 CARACTERÍSTICAS DE SISTEMAS DISTRIBUÍDOS

Depois de termos analisado várias definições e vários tipos de sistemas distribuídos, parece-nos interessante escolher algumas das características mais importantes destes sistemas para serem estudadas com mais detalhe. Foram escolhidas as seguintes características: transparência, flexibilidade, modularidade, confiabilidade e tolerância a falhas.

II.4.1 TRANSPARÊNCIA

No Workshop entitulado "Operating Systems in Computer Networks", já mencionado anteriormente [161], teve uma seção específica sobre este tema. Foi colocado por KREISSIG a necessidade de ter vários graus de transparência em uma rede, para obter um **sistema distribuído homogêneo**. O conceito de homogeneidade aqui se refere a homogeneidade lógica e não física, como foi definida anteriormente. Para que um sistema fisicamente heterogêneo, isto é, composto por um conjunto interligado de

computadores de diferentes tamanhos, desde estações de trabalho até computadores grandes do tipo "mainframe", se apresente ao usuário como um sistema homogêneo, tão simples de usar quanto um computador único, é necessário implementar transparência total.

Esta transparência reduz enormemente o custo do desenvolvimento e manutenção de software. Permite também uma grande flexibilidade para a configuração do sistema, incluindo estações de trabalho sem disco, compartilhamento transparente de periféricos, crescimento incremental em relação a diferentes configurações, sem produzir nenhum efeito virtual no software de aplicação necessário para explorar ao máximo a configuração modificada.

Um exemplo deste tipo de filosofia de transparência virtual completa é o sistema Locus [60, 82], que, a partir de um projeto de pesquisa sobre computação distribuida, evoluiu para um sistema UNIX distribuido, transparente e completamente compatível com o UNIX padrão. No sistema Locus, programas que rodam numa única máquina com o sistema UNIX, podem rodar sem modificações com recursos distribuidos. Isto é, podem ser escolhidos programas aplicativos já desenvolvidos, e, definindo alguns poucos parâmetros, de configuração, colocados para executar com uma parte carregada numa máquina, outras partes em outras máquinas, tendo acesso a dados e dispositivos distribuidos através da rede Locus, sem precisar modificar a programação. Esta filosofia permite o aproveitamento no ambiente distribuido de uma grande coleção de software de aplicação, o que representa um pre-requisito necessário para o uso comercial de computação distribuida.

Em [161] KREISSIG distingue entre **transparência de acesso**, onde o processo tem o mesmo tipo de mecanismo de acesso para recursos locais e remotos; **transparência de endereço**, onde o endereço de um recurso é invisível ao método de acesso; **transparência de controle**, onde toda a informação que descreve o sistema tem aparência idêntica para o usuário ou para a aplicação; e **transparência de execução**, que permite balanceamento de carga, transporte dos programas para onde estão os dados, comunicação entre processos que estão em processadores diferentes, etc. Associado a cada nível de transparência existe um conjunto de funções, que é um subconjunto das funções do nível

anterior.

A transparência de acesso e a de endereço estão ligadas ao problema de nomeação e de ligação. KLINE em [82] define **transparência de nomeação** como a habilidade de ter acesso a um recurso pelo mesmo nome a partir de qualquer ponto da rede, e para isto cada recurso deve ter um nome global, único na rede. Esta propriedade permite que os programas sejam transportados dinamicamente na rede e funcionem corretamente em qualquer endereço. Em [62] GRAY coloca que a condição, para que haja transparência de endereço, é que o endereço de um recurso não possa ser deduzido à partir do seu nome. Por outro lado, em [82] a transparência de endereço é definida como a habilidade de trasladar a posição de um arquivo sem precisar mudar seu nome. Para os programas que contêm nomes de arquivos embutidos neles, a falta de transparência de endereço implicaria que os programas teriam que ser modificados e recompilados quando um recurso fosse trasladado.

Vejamos agora o problema de ligação dos identificadores associados aos objetos, como foi colocado em [127] por PETERSON. Os objetos são geralmente referenciados por um sistema único de identificadores. O identificador será ligado a um determinado endereço físico para permitir acesso ao objeto físico. Um aspecto importante destas ligações é a frequência com que são feitas, e o conjunto de ações que podem ser executadas entre o estabelecimento de uma ligação e seu término posterior. Se o objeto não puder ser trasladado depois de ser ligado a um identificador, então esse identificador pode conter o endereço e a informação de roteamento. Por outro lado, se o objeto puder ser trasladado, então o identificador deve ser independente do endereço, e os acessos locais e remotos devem ser transparentes.

Num sistema distribuido, existem vários níveis nos quais podemos escolher construir o suporte de transparência. Por exemplo, pode-se definir um nível acima do sistema operacional, no qual serão implementadas operações distribuidas e transparência, ou podemos fazê-lo mesmo em níveis mais altos, tais como no nível da linguagem de programação ou no nível de banco de dados. Quanto mais alto é o nível, mais difícil se torna a implementação destas facilidades. Em Locus foi escolhida a implementação no nível mais baixo, isto é, no nível do núcleo, de

maneira de obter um sistema operacional totalmente transparente.

Existe ainda o conceito de **transparência de desempenho** [161, 62] que está relacionado com o fato de que os usuários de um sistema não deveriam experimentar uma degradação excessiva de desempenho nos acessos remotos.

No final da seção sobre Sistemas Operacionais Distribuídos Homogêneos, do Workshop já mencionado [161], foi colocada, por CHERITON, a pergunta mais controvertida: "Porque homogeneidade?" Será que é necessária uma transparência completa da rede, particularmente num ambiente heterogêneo, ou será que é melhor que o usuário seja ciente das diferenças de capacidades dos recursos ligados na rede? Na discussão não houve consenso e os participantes se dividiram em dois grupos, um a favor de homogeneidade e o outro contra.

Podemos concluir que a transparência implica num custo que as vezes não é desejável pagar, particularmente quando é fundamental um bom desempenho do sistema. Neste caso é mais importante ter acesso explícito aos recursos da arquitetura distribuída para otimizar seu uso. Por outro lado, para sistemas de uso geral, por exemplo, podem ser mais importantes as propriedades de homogeneidade e transparência do sistema.

II.4.2 FLEXIBILIDADE

Flexibilidade pode ser definida como a propriedade que permite que um sistema se adapte às necessidades de mudanças.

Um sistema distribuído geralmente é um sistema grande de vida média longa, portanto sujeito a modificações tanto em relação a novas tecnologias a serem incorporadas, quanto em relação a mudanças do ambiente da aplicação.

É portanto necessário que o sistema permita expansão, substituição de recursos mais poderosos e modernos, e mudanças de algumas funções da aplicação em resposta a novas necessidades; podemos chamar estas mudanças do **tipo evolucionário**.

Pode surgir também a necessidade de realizar mudanças do **tipo operacional**; estas estão ligadas ao funcionamento do sistema, como, por exemplo, o aparecimento de falhas do sistema, que exigem sua reestruturação para isolá-las.

Estes dois tipos de mudanças em alguns casos requerem

medidas parecidas a serem tomadas, mas a diferença principal entre os dois é o fato que as mudanças do tipo evolucionário não podem ser previstas na hora de projetar o sistemas, e portanto só podem ser tratadas adicionando novas funções ao sistema.

Um conceito importante ligado à flexibilidade é a autonomia, colocada por PETERSON em [127] e já mencionada anteriormente. Os componentes de um sistema distribuido devem ser autônomos, isto é, o sistema deve satisfazer o requisito de poder sofrer o desligamento de um processador da rede sem ser afetado o seu funcionamento geral. Podemos expressar ainda estes conceitos dizendo que a organização e o funcionamento de cada nó devem ser independentes dos outros nós da rede.

Em determinados casos não é possível parar o sistema como um todo para introduzir algumas modificações, seja por razões econômicas ou de segurança. Nesses casos é então necessário fazer as modificações de forma dinâmica.

Os tipos de flexibilidade de um sistema distribuido podem ser caracterizados da seguinte forma, como foi colocado em [104] por MAGEE:

Flexibilidade Funcional: é a habilidade de modificar a funcionalidade de um sistema, seja acrescentando novas funções ou substituindo as já existentes. Um elemento importante da flexibilidade funcional é a facilidade com que podem ser determinadas as implicações de uma mudança, por exemplo, quais as partes do sistema afetadas pela mudança.

Flexibilidade de Implementação: é a habilidade de modificar a forma na qual uma função é implementada. Esta mudança pode ser necessária no caso de precisar melhorar o desempenho do sistema, por exemplo, implementando em hardware uma função que estava implementada antes em software, ou mudar os algoritmos de uma função por outros mais eficientes, ou fazer modificações tecnológicas, por exemplo, substituir processadores obsoletos.

Flexibilidade de Topologia: é a habilidade de mudar a estrutura, tanto lógica quanto física, de um sistema. A estrutura física é definida pela localização física dos

elementos de computação e as suas interligações através de elos de comunicação. A estrutura de software é definida pela localização dos componentes de software nos computadores físicos e as ligações lógicas entre os componentes.

Flexibilidade de Domínio de Tempo: é a habilidade de introduzir uma mudança no sistema em qualquer ponto no tempo. Uma medida da flexibilidade de domínio de tempo é quantas partes do sistema devem ser paradas ou desligadas para poder implementar a mudança.

Sistemas frouxamente ligados, isto é, com nós mais autônomos comparados com os nós de sistemas fortemente ligados, apresentam uma flexibilidade alta. É interessante salientar que, quanto maior a transparência de um sistema, maior será a sua flexibilidade, já que os mecanismos que implementam a transparência tomarão conta das mudanças possíveis, sem que o usuário do sistema tome conhecimento. A modularidade, que será analisada à continuação, é uma característica necessária para a implementação de flexibilidade, já que as mudanças precisam ser feitas em módulos isolados de maneira que as interfaces sejam mantidas iguais e não ocorram alterações nos outros módulos do sistema que não estejam envolvidos nas mudanças.

II.4.3 MODULARIDADE

Como foi colocado em [142] por SEGRE e STANTON, os programas distribuidos têm em geral a característica de serem programas grandes, tanto os que representem software básico para arquiteturas distribuidas, como a maioria das aplicações.

Entre as propriedades que estes sistemas de software precisam ter, podemos destacar as seguintes: o sistema deve funcionar corretamente, deve ter uma vida longa e deve ser facilmente modificável. Para gerenciar o projeto e a produção de sistemas grandes é necessário dispor de metodologias e ferramentas adequadas para poder controlar a sua complexidade.

O requisito principal é que os sistemas grandes sejam constituidos de componentes individuais chamados **módulos**, como

foi colocado por PARNAS em [124]. Estes sistemas serão geralmente desenvolvidos por um grupo de pessoas que programarão separadamento os módulos individuais, escritos até em linguagens diferentes. O requisito de modularidade está associado à independência e autonomia, já que cada módulo será entendido e implementado independentemente dos outros módulos do sistema. A tarefa atribuida a cada pessoa deve estar definida claramente e sem ambiguidade, de forma que ela não precise conhecer o trabalho dos outros. Para poder programar um módulo independentemente dos outros, precisa-se definir cuidadosamente as interfaces entre os módulos.

Esta modularidade é fundamental para provar a correção do sistema que será baseada na correção dos módulos que o constituem, garantindo que não será afetada a correção da cooperação entre módulos.

Estes sistemas grandes são geralmente caros, e é portanto conveniente que partes deles possam ser reusadas. A estruturação modular oferece a possibilidade de que os módulos, desenvolvidos independentemente, sejam guardados em bibliotecas para depois serem ligados para construir um único programa. Isto permite reusar módulos já existentes de outros sistemas.

Devido à longa vida destes sistemas, é importante que eles possam ser facilmente modificáveis. A funcionalidade dos módulos, escondendo os detalhes de implementação, permite que sejam feitas modificações internas sem ocasionar mudanças no sistema todo.

Uma das ferramentas principais para o projeto de sistemas grandes é o uso de linguagens de programação de alto nível, que deve ser compatível com a metodologia de software utilizada no projeto de software. Estas linguagens facilitam a escrita de software, permitindo uma maior rapidez e evitando erros que poderiam acontecer usando linguagens de baixo nível. Existem ainda linguagens concorrentes que permitem a deteção de erros de concorrência na etapa de compilação, e que permitem obter sistemas mais confiáveis. É importante destacar a forte influência originada pelo projeto modular de software sobre as linguagens de programação.

As linguagens devem oferecer ferramentas para controle de estruturas em alto nível, para projeto descendente ("top-down"), para programação estruturada, para definir módulos, para esconder

informação e definir dados abstratos, para garantir confiabilidade através de tratamento de exceções, e para estruturar uma coleção de módulos num único sistema.

O projeto descendente para aplicações grandes deve dar suporte à possibilidade de decompor o sistema em módulos. Seguindo o princípio de esconder informação, o projetista deve distinguir claramente entre o que o módulo faz, chamado geralmente de interface do módulo (os objetos que o módulo exporta para serem utilizados por outros módulos), e como ele faz, ou seja a sua implementação (seus detalhes internos). A interface do módulo deve especificar claramente as entidades internamente definidas que são exportadas para serem usadas por outros módulos, e as entidades externamente definidas importadas a partir de outros módulos. Projetar um módulo consiste em projetar a sua interface e a de outros módulos que serão utilizados por ele.

O módulo é efetivamente uma cerca em volta de um grupo de declarações (tipos, variáveis, procedimentos, etc) estabelecendo um escopo para os identificadores. Podemos considerar a cerca como uma parede impenetrável: os objetos declarados fora do módulo são invisíveis dentro dele, e os declarados dentro são invisíveis fora. Esta parede é furada seletivamente por duas listas de identificadores: a lista de importações e a lista de exportações.

O módulo, agrupando declarações de objetos e selecionando alguns deles para serem externamente visíveis, é apropriado para definir tipos abstratos de dados dos quais podem ser criadas várias instâncias.

Sistemas grandes são construidos em níveis usando uma **estrutura hierárquica**. O grande número de componentes elementares demanda um número grande de níveis lógicos, cada um descrito por um conjunto de abstrações. É então desejável especificar interfaces e poder separar as especificações de um programa das especificações das interfaces. Esta separação apresenta a vantagem de poder divulgar a especificação das interfaces, enquanto as implementações das interfaces são particulares de seus implementadores. Este esquema é particularmente útil quando embutido na linguagem de programação, já que seu compilador poderá testar a consistência das partes individuais do programa,

isto é, poderá compilar separadamente os módulos. O compilador deve verificar que cada módulo é consistente com a especificação da sua interface e que as chamadas para um outro módulo M são consistentes com a interface de M. Somente desta maneira é possível determinar se as mudanças de um módulo podem ser implementadas sem implicar em mudanças em nenhum de seus usuários; a condição para isto é que a sua interface não seja modificada.

Como foi colocado por WIRTH em [177] o conceito de módulo foi evoluindo. Inicialmente um módulo era definido com o objetivo de juntar todos os procedimentos que compartilham um conjunto de variáveis, de maneira que estes procedimentos sejam os únicos que possam ter acesso exclusivo a essas variáveis; esta é a idéia básica do conceito de monitor introduzida por HOARE [67]. Posteriormente o conceito de módulo evoluiu focalizando mais a estrutura de dados contida nele; isto é, o módulo ficou caracterizado mais pelos dados que ele contém, e às vezes esconde, do que pelo conjunto de procedimentos que ele exporta. Desta forma, o módulo passou a ser considerado um tipo abstrato de dado.

O reconhecimento de uma modularização útil da estrutura de dados envolvida é a chave para encontrar uma decomposição apropriada de um programa em módulos, como PARNAS muito bem mostrou em [124]. Programar de forma modular é bem mais difícil, já que força o programador a raciocinar desde o início mais cuidadosamente sobre a estrutura do programa. No entanto, o benefício obtido é que, uma vez projetado o programa, será mais fácil modificá-lo, documentá-lo e entendê-lo. Para sistemas grandes, que geralmente têm uma vida longa, o ganho em manutenção justifica o esforço investido no projeto. Mais ainda, em vários casos achar a estrutura correta no início não é só um benefício mas é uma questão vital.

Outro ponto importante a ser levantado é como o conceito de módulo permite encapsular o uso de facilidades dependentes de máquina, as quais, em geral, desejamos esconder de outras partes do programa. Estas facilidades são necessárias para programar, por exemplo, gerenciadores de recursos físicos. Isto permite que todos os níveis de um sistema possam ser expressos numa única e mesma linguagem, sem ser muito restritivo nos módulos de baixo

nível, nem sendo permissivo demais nos módulos de alto nível. Um exemplo desta característica é o uso de Módula-2, que oferece ferramentas para programar gerenciadores de dispositivos e de interrupções ao nível do hardware, como é ilustrado por WIRTH em [180].

Como já foi mencionado anteriormente o conceito de modularidade está fortemente ligado às outras características escolhidas, particularmente à flexibilidade, confiabilidade e tolerância a falhas. Na seção que trata da estruturação modular de programas distribuídos serão analisados outros aspectos relacionados com modularidade.

II.4.4 CONFIABILIDADE E TOLERÂNCIA A FALHAS

A preocupação de incorporar meios para tolerar falhas, com o objetivo de aumentar a confiabilidade de um sistema de computação, ficou estabelecida desde os trabalhos originais de von Neumann na década dos 50. Com o avanço da tecnologia, as técnicas utilizadas têm evoluido muito, para satisfazer os requisitos de confiabilidade, cada dia maiores. À medida que os computadores sejam utilizados cada vez mais em tarefas altamente críticas, a necessidade de tolerar falhas aumentou, e se fez sentir desde as áreas militares e aeroespaciais até os ambientes mais gerais da indústria, do comércio e da saúde.

Originalmente as técnicas de tolerância a falhas eram desenvolvidas para resolver problemas de mal funcionamento previsíveis de componentes de hardware, como é colocado por ANDERSON e LEE em [3]. À medida que os sistemas foram evoluindo e ficando cada vez mais complexos, particularmente por causa do software, surgiu a necessidade de um novo enfoque que incluisse ambos o hardware e o software. Existe entretanto a preocupação em relação à inclusão das técnicas de tolerância a falhas, porque acrescentando mais complexidade ainda ao sistema, pode-se aumentar a possibilidade de ocorrência de falhas. Isto leva à necessidade de pesquisar técnicas seguras para garantir que a confiabilidade do sistema seja aumentada e não diminuida.

Por outro lado, podemos notar que o aumento da complexidade do hardware através do uso da tecnologia de VLSI, diminui consideravelmente a ocorrência de falhas no nível do hardware, e

portanto a importância de técnicas de tolerância a falhas para o software cresce cada dia mais.

Identificamos duas direções, nas quais as técnicas têm evoluido. Uma é a necessidade de **prevenir as falhas**, evitando a sua ocorrência. A outra, supondo que os sistemas nunca serão perfeitos, deverá incluir tratamento para as **falhas não previsíveis**. Este último enfoque é o mais difícil de ser tratado, e é onde se concentram as pesquisas para achar mecanismos que garantam a robustez dos sistemas.

Existem muitos trabalhos sobre esta área na literatura, mas não sendo um ponto central do objetivo do nosso estudo, abordaremos somente alguns temas que estão em discussão, sem entrar em detalhes das técnicas existentes. Queremos salientar que a confiabilidade de um sistema é uma característica fundamental, e que como apontaremos nas conclusões, é um dos pontos essenciais a ser desenvolvido e acrescentado no futuro, à nossa proposta de ambiente de programação distribuida.

Ao nível de linguagens de programação podemos citar principalmente quatro linguagens que incluiram mecanismos para o tratamento de falhas: ADA (ICHBIAH [73]), MESA (MITCHELL et alii [113]), CLU (LISKOV et alii [95]), e, em particular para sistemas distribuidos, ARGUS (LISKOV [99, 98, 97]), que é uma extensão de CLU.

A estruturação modular de software facilita a prevenção de falhas, já que permite o desenvolvimento e a verificação de módulos de forma independente. Para que os erros ocorridos durante a execução não ocasionem a paralização total do programa, é necessário ter mecanismos para interceptação de situações de exceção, e a consequente tomada de providências corretivas, entre as quais pode ser considerada a reconfiguração dinâmica de software.

A maioria das linguagens, que provêem um mecanismo de tratamento de falhas, implementam o seguinte modelo. Chamamos operações os procedimentos exportados por um módulo. Cada operação é programada para executar alguma computação sobre dados de entrada no seu domínio de definição. Estes dados são caracterizados pela afirmativa de caso normal de entrada da operação, e a aplicação da operação a esses dados deverá gerar um resultado que satisfaça a afirmativa de saída prescrita. A

operação deve verificar a afirmativa de caso normal de entrada e avisar ao usuário no caso desta não ser satisfeita; este caso é chamado de **exceção** da operação e dizemos que a operação a assinala ao usuário. Ao ocorrer uma exceção, o controle é então passado para uma rotina chamada **tratador de exceção**. Dependendo da estrutura da linguagem e da possibilidade de aninhamento de módulos, as **exceções podem ser propagadas**, para serem atendidas por tratadores de exceção incluídos em outros módulos de mais alto nível. As propagações devem satisfazer as regras de escopo do aninhamento dos módulos.

Este modelo, no entanto, levanta uma discussão em relação ao **acoplamento entre o usuário e a operação**. Para satisfazer os princípios de modularidade deve-se diminuir o grau de acoplamento citado. Para isto é mais adequado que o tratador de exceção seja definido pelo usuário. Este modelo alternativo foi assumido por YEMINI e BERRY em [182], que defendem o princípio de que é o usuário que sabe o que precisa ser feito no caso de ocorrer uma exceção na chamada de uma operação, e questionam a existência de um tratador geral associado à operação. Em [182] é apresentada uma série de requisitos que devem ser satisfeitos por um mecanismo de tratamento de exceções, chamado de **modelo de substituição**, para permitir uma resposta flexível por parte dos usuários à deteção de exceções, sem comprometer a modularidade. Do conjunto de requisitos destacamos: respostas alternativas dos tratadores, parametrização dos tratadores, a propagação explícita de exceções e o papel verificador do compilador.

Não existe consenso em relação a um modelo em particular, e a segunda alternativa apresentada, que coloca o tratador de exceções no usuário, é questionada pelos autores que defendem a importância da integridade de tipos abstratos de dados. Estes seriam compostos pelas suas estruturas de dados, as operações associadas e os tratadores de exceções ligados a cada operação.

O problema de confiabilidade é ainda mais crítico em **sistemas distribuídos**. Por um lado, pelo fato de sistemas distribuídos possuirem múltiplas partes de hardware e software funcionando conjuntamente, as chances de uma dessas partes falhar é bem maior do que ocorre num sistema simples. Um sistema distribuído pode apresentar falhas de vários tipos, dentre as quais, aquelas provenientes de defeitos nas linhas de comunicação

que interligam os processadores, e as referentes a colapsos nos processadores ou nos dispositivos periféricos. Para que um sistema distribuido seja confiável e robusto, é necessário que ele possa continuar funcionando corretamente, mesmo com a presença de certos tipos de defeitos ou interferências indevidas.

Por outro lado, os sistemas distribuidos possuem características que facilitam a inclusão de mecanismos de tolerância a falhas. O acoplamento geralmente frouxo dos processadores no sistema ajuda a isolar o efeito das falhas. A possibilidade de adicionar redundância dos componentes, tanto de hardware quanto de software, é utilizada para dar suporte à implementação de políticas de reconfiguração para tratar as falhas. Componentes críticos e técnicas de tratamento de tolerância a falhas podem ser utilizadas para mascarar as exceções.

Existe outra discussão que gostaríamos de colocar em relação à **inclusão de mecanismos de tolerância a falhas nas linguagens de programação.**

ANDREWS e OLSSON apontam em [6] que as **falhas de hardware** podem ser tratadas no contexto de uma linguagem para programação distribuida, de três maneiras diferentes. Por um lado, podem ser providos mecanismos na linguagem de alto nível, que mascarem a ocorrência de falhas, ou que as tratem para obter alguma forma de recuperação. Exemplos deste enfoque são as ações atômicas implementadas por LISKOV [97], as ações de tolerância a falhas de SCHLICHTING e SCHNEIDER [139] e as chamadas de procedimentos reiteradas, implementadas por COOPER [43]. Estas técnicas são úteis, especialmente para programas aplicativos. Entretanto, cada uma delas precisa de um núcleo ("kernel") extenso para dar suporte em tempo de execução, que na prática é um sistema operacional de propósito especial. Portanto, estas técnicas são de muito alto nível para serem incorporadas em linguagens para programação de sistemas.

No extremo oposto, existe outro enfoque no qual a linguagem provê somente um mecanismo de saída por temporização ("timeout"), para evitar que um processo fique bloqueado externamente esperando por uma chamada de operação terminar. Este enfoque é utilizado em linguagens tais como ADA (comando "delay")(ICHBIAH [73]) e SR (ANDREWS [4]).

O terceiro enfoque intermediário, apresentado por ANDREWS e OLSSON em [6], foi adotado por eles para a versão estendida de SR [7]. O núcleo é o encarregado de detectar a falha, através de um mecanismo que na prática é uma saída por temporização, mas o programador é o responsável pelo tratamento da falha, assim que for detectada e assinalada. Por exemplo, se um rescurso detectar que outro falhou, pode ser apropriado criar uma nova instância deste recurso.

Em relação às **falhas de software**, eles concordam com BLACK [22], que estas não precisam ser consideradas, e as únicas exceções que deveriam ser tratadas são aquelas derivadas das limitações do hardware no qual o programa é executado. Eles reconhecem que a sua posição é muito radical, mas acham que só o tempo dirá se estão certos.

Continuando este raciocínio gostaríamos de colocar as opiniões de LOQUES e KRAMER [103]. Eles consideram que em geral **as técnicas de tolerância a falhas são caras**, e portanto é aconselhável utilizá-las de forma seletiva e localizada de acordo com os seguintes critérios:

i) somente os módulos que precisam de tolerância a falhas devem pagar por isto.

ii) não devem ser impostas técnicas especiais de programação ou restrições.

iii) é necessário dispor de diferentes técnicas de tolerância a falhas para satisfazer diferentes requisitos de confiabilidade.

Eles acrescentam que é fortemente desejável que haja uma **independência entre a programação e a tolerância a falhas**. É necessário projetar e testar o comportamente lógico de um sistema de aplicação sem se preocupar com este aspecto. Numa segunda fase do projeto, deve ser possível acrescentar mecanismos específicos de tolerância a falhas onde for necessário, mas sem ter que reprojetar os módulos de aplicação. Isto permite o uso de técnicas e ferramentas padrão para diferentes requisitos de confiabilidade. Esta transparência permite ainda reusar módulos de programas, facilitando a construção modular de sistemas de

aplicação. Portanto, para obter transparência é necessário que os mecanismos de tolerância a falhas sejam ortogonais às funções a aplicação.

Como vemos, esta área está ainda em aberto e precisa de estudos mais aprofundados. Podemos notar também o forte relacionamento que existe com outros temas que serão discutidos neste trabalho, tais como: modularidade, separação de funções do ambiente de programação distribuida, reconfiguração dinâmica, problemas de substituição, transparência, etc.

II.5 ESTRUTURAÇÃO MODULAR DE PROGRAMAS DISTRIBUIDOS

O processo de desenvolvimento de software pode ser dividido em várias etapas, como foi mostrado por GHEZZI em [59] e MONTANARI em [114], que geralmente serão sequenciais, a menos que sejam encontrados alguns erros ou modificados alguns requisitos. Neste caso será necessário repetir alguma fase ou grupos de fases.

A forma mais encontrada na literatura de esquematizar essas fases é a seguinte:

- análise de requisitos e especificação
- projeto de software e decomposição
- implementação (codificação)
- certificação
- manutenção

O trabalho em cada uma destas fases de desenvolvimento de software pode ter suporte de ferramentas automatizadas.

Estas ferramentas estão evoluindo gradativamente, na medida que cada dia mais é reconhecida a necessidade de automação, para aumentar a produtividade de programação reduzindo a ocorrência de erros. A fase que atualmente tem melhor suporte é a fase de codificação com ferramentas tais como editores de texto, compiladores, ligadores e bibliotecas.

Como já foi dito, o desenvolvimento de software envolve muito mais do que codificação. Em [59] GHEZZI define o conceito de ambiente de desenvolvimento de software como o conjunto integrado de ferramentas e técnicas que auxiliam no

desenvolvimento de software. É importante que estas ferramentas não só funcionem bem no seu conjunto, mas também que sejam compatíveis com as ferramentas utilizadas em outras fases. Por exemplo, a linguagem de programação deve ser compatível com a metodologia de projeto, isto é, com as ferramentas que dão suporte na fase de projeto.

Os pontos levantados aqui fazem parte da disciplina de engenharia de software que se encarrega de desenvolver as ferramentas para cada fase do ciclo de vida do software. O gerenciamento do projeto e da produção de sistemas grandes, como é o caso de sistemas distribuídos, requer metodologias e ferramentas adequadas para poder manter o controle da complexidade.

Aqui estamos particularmente interessados em analisar o relacionamento entre as linguagens de programação e as metodologias para projetos de software. As primeiras linguagens de programação foram projetadas para codificação, mais do que para desenvolvimento de software. WIRTH coloca em [177] o seu enfoque em relação a linguagens de programação em oposição ao mais habitual, que considera as linguagens como um meio de comunicação entre o homem e a máquina. Para WIRTH, as linguagens de programação são conjuntos de ferramentas abstratas para a construção do software que complementa o hardware incompleto; cada programa estende o hardware criando uma máquina que se comportará da forma desejada.

As áreas de metodologia de projeto de software e a de projeto de linguagens de programação podem convergir, como de fato acontece em alguns casos. O exemplo mais claro nesta direção é o princípio de esconder informação (como metodologia de projeto) e abstração de dados (como princípio de projeto de linguagem). O objetivo do projeto modular de software exerce uma influência forte sobre as linguagens de programação. Podemos encontrar na literatura diferentes enfoques em relação às ferramentas utilizadas.

Para MONTANARI [114] é importante basear o ciclo inteiro de vida de software numa única linguagem. Ele considera que os aspectos de desenvolvimento descendente são garantidos se for permitida a especificação, no lugar da implementação, de algumas partes do programa. A linguagem de especificação pode ser uma

linguagem assertiva incluída na linguagem de programação e/ou um cálculo baseado em axiomas que dariam as propriedades das funções e dos procedimentos que estariam faltando. As várias linguagens utilizadas num ambiente de desenvolvimento de software devem ser todas semanticamente consistentes, isto é, elas devem representar todos os aspectos de uma mesma linguagem.

O ambiente deve ser capaz de dar suporte ao projeto durante todas as fases do ciclo de vida de software. Ele deve ter duas propriedades aparentemente contraditórias: ele deve ser integrado e granular. Integração significa que as ferramentas devem ser consistentes e devem combinar naturalmente, enquanto granularidade significa que as ferramentas devem ser pequenas e ortogonais. Para resolver esta dualidade, pode-se aplicar os princípios de programação modular, isto é, interfaces claras e funcionalidade simples. Neste contexto, definir as interfaces significa especificar as linguagens de programação e as representações internas do programa. Portanto usar uma única linguagem de programação simplifica as coisas.

DE REMER e KRON [47] apresentam um enfoque diferente em relação ao uso de linguagens de programação para a produção de software. Eles ponderam que o ciclo de vida de software no caso de sistemas grandes pode ser bem diferente comparado com o de programas pequenos, e argumentam que estruturar uma coleção de módulos para formar um sistema (programação em larga escala) é uma atividade intelectual essencialmente distinta e diferente da de construir módulos individuais (programação em pequena escala). Eles concluem portanto que é necessário utilizar linguagens distintas para as duas atividades, que serão chamadas aqui de **linguagem de programação em larga escala** (LPL) e **linguagem de programação em pequena escala** (LPP). É colocado também que o uso de uma LPL separada facilita o uso de módulos escritos em linguagens diferentes.

Na literatura achamos também outras formas de denominação para as linguagens de programação em larga escala. Em [47] a linguagem utilizada para descrever a estrutura de um sistema é chamada de **linguagem de interconexão modular** ("Module Interconnection Language", MIL) e algumas delas serão tratadas posteriormente no Capítulo V. Estas linguagens precisam conter as primitivas necessárias para o gerenciamento dos módulos e para

testar a consistência de suas interconexões. As partes do programa que especificam operações sobre módulos devem ser elas mesmas módulos, de forma que módulos cada vez maiores possam ser facilmente construídos. Vemos então que estas linguagens permitem definir a configuração de um sistema e são às vezes denominadas de **linguagens de configuração.**

Como foi colocado por SEGRE e STANTON em [142] a **configuração** de um sistema distribuído depende de três aspectos principais:

- especificação de quais componentes comporão o programa;
- especificação das **ligações de comunicações** entre os componentes;
- especificação da **localização** dos componentes.

Evidentemente o primeiro requisito que precisa ser satisfeito é que seja possível o desenvolvimento e compilação em separado dos módulos. Como foi mostrado na seção sobre modularidade, é necessário que a LPP tenha condições de definir adequadamente as interfaces entre os diferentes módulos. Estas interfaces geralmente ou são portas de mensagens, conceito que será amplamente discutido no próximo capítulo (*MOD (COOKO [42]), Port Directed Communication (SILBERSCHATZ [149]), CONIC (KRAMER et alii [86])). ou são procedimentos, (ADA (ICHBIAH et alii [73]), "Distributed Processes", DP (BRINCH HANSEN [29]), MODULA (WIRTH [176]), Modula-2 (WIRTH [178]), "Sinchronization Resources", SR (ANDREWS [4]), MESA (LAMPSON e REDELL [76], MITCHELL et alii [113]), e como tais estão associadas aos tipos de mensagem ou de parâmetros e resultados que manuseiam.

A especificação de interfaces é facilitada em muitas das LPPs correntes (ADA (ICHBIAH et alii [73]), CONIC (KRAMER et alii [86]), MESA (MITCHELL et alii [113]), Modula-2 (WIRTH [178]), SR (ANDREWS [4])) por mecanismos de importação e exportação de listas de declarações, com sintaxes diversas. Uma unidade de compilação ou módulo, importa as interfaces externas que necessita para comunicação, e por sua vez pode exportar sua inferface para que possa ser usada por outros módulos. No caso de comunicação através de portas, todos os módulos usuários de um determinado tipo de porta devem importar sua definição. Um

ambiente de desenvolvimento de programas inclui um sistema de arquivos onde são guardadas as interfaces, normalmente numa forma gerada a partir da compilação do código fonte do módulo exportador. Observa-se que esta estrutura impõe uma ordenação parcial da sequência de compilação dos módulos.

Para garantir a integridade dos tipos de dados usados nos módulos, já verificada individualmente para cada módulo pelo compilador, precisamos de mecanismos na implementação da LPL para verificar que dois módulos que serão combinados adotarão a mesma definição de sua interface comum, isto é, podemos considerar que na montagem de uma configuração, o configurador é o responsável por testar a interface de cada módulo em relação aos recursos providos por outros módulos e especificados nas próprias interfaces.

O estabelecimento das ligações de comunicação entre os módulos mostra uma certa diversidade de modelos. Em alguns casos, a estrutura de comunicação coincide com a estrutura modular do programa, sendo portanto estática, como é o caso de CONIC. Outras linguagens permitem o estabelecimento dinâmico de ligações de comunicação através da passagem, em mensagens ou como parâmetros, dos endereços de outras portas ou conjuntos de procedimentos que representam a interface do módulo. Na linguagem MCD (RUGGIERO e BRESSAN [137]) um endereço para respostas é passado junto com as mensagens. Em Ada (ICHBIAH et alii [73]), tarefas podem ser passadas como parâmetros, permitindo acesso às interfaces de comunicação que estas exportam. Tanto em ARGUS (LISKOV [98]) quanto em SR (ANDREWS [5]), as operações são invocadas através de manuseadores ("handlers") e direitos de acesso ("capabilities"), respectivamente, que podem ser passados nas mensagens para render as operações acessíveis a outros módulos (guardiães em ARGUS e recursos em SR). Neste caso a ligação entre o parâmetro e a interface remota é estabelecida **implicitamente** por mecanismos da LPP. Em linguagens que utilizam portas, antes destas serem usadas para comunicação, precisamos ligar explicitamente pares de portas, e a LPP possui primitivas ou comandos para efetuar as ligações [137, 54]. O projeto CONIC [104, 152, 105, 106, 87] tem evoluido incluindo a possibilidade de definir configuração dinâmica. Para isto foram acrescentadas à linguagem de configuração, primitivas de destruição dinâmica de processos, de

desligamento de portas e comandos para elaborar as mudanças, que serão processados por um gerenciador de configuração.

De maneira geral, ligações dinâmicas estão associadas a linguagens que prevêem a criação e destruição de processos, enquanto ligações estáticas estão associadas à estruturas estáticas de concorrência.

Em arquiteturas distribuídas, além de determinar quais módulos comporão um programa, temos a liberdade de dizer onde serão carregados estes módulos, uma vez que existem várias alternativas. Precisamos portanto ter uma nomenclatura para os nós físicos, e um mapeamento dos módulos para estes nós. Observa-se que somente é importante para o programador poder especificar este mapeamento no caso de haver recursos não uniformemente distribuídos nos nós, como é o caso num sistema distribuído de controle de processos, ou por causa da topologia de comunicação entre os nós físicos. A implementação de CONIC [86] permite especificar a localização de módulos para controle de processos, através do comando de instanciação destes módulos em nós físicos específicos.

As características de programação em larga escala que descrevemos requerem suporte linguístico para sua realização. CONIC, por exemplo, mostra uma separação quase completa entre a LPP e a LPL. (A LPP consulta arquivos globais para obter definições de tipos de portas). MCD (RUGGIERO e BRESSAN [137]) inclui na sua LPP todas as características mencionadas, mas devemos notar que esta linguagem não prevê (ainda) compilação em separado. ADA e MESA dependem de dois níveis de linguagem que se comunicam através de uma base de dados (incluída no APSE) no caso de ADA, e através de um sistema de arquivos no caso de MESA.

Um caso muito interessante é o ambiente de programação no sistema UNIX (KERNIGHAN e MASHEY [80]), onde os programas são escritos na linguagem C, estendida com rotinas de suporte de entrada/saída e programação concorrente do sistema operacional. Através da linguagem SHELL (uma LPL) podem ser combinados programas em C, interligados através de dutos ("pipes"). Agora o interpretador de SHELL é meramente outro programa não privilegiado escrito em C, portanto todas suas facilidades estão ao alcance de qualquer programa. Temos então a clara alternativa de realizar as funções da LPL através da LPP.

Outro caso que merece consideração especial é o de nós frouxamente acoplados, por exemplo, os nós que correspondem a centros autônomos de computação, interligados através de uma rede de comunicação à distância. Em geral estes nós terão características diferentes, sendo padronizados os protocolos de comunicação e talvez a linguagem de programação (LPP) em cada nó. Para estabelecer ligações entre módulos em nós diferentes, no caso de não existir um só interpretador de uma LPL por causa da autonomia dos nós, a solução é padronizar primitivas de programação em larga escala para serem usadas ao nível da LPP (ADA). No projeto Cnet (FANTECHI et alii [54]), foram definidas primitivas para criar portas e para ligá-las com portas remotas. Evidentemente, cada nó terá que implementar estas primitivas localmente.

Podemos então notar que existem várias linguagens adotando os dois enfoques diferentes, isto é, linguagens distintas em dois níveis, bem separados, LPP e LPL, e as que juntam as características de cada uma, numa só, sendo consideradas uma como extensão da outra. De certo modo esta discussão lembra o debate sobre a necessidade de ter uma linguagem de controle distinta da linguagem de programação, colocado por UNGER em [169]. Neste último contexto, a solução mais comum é a separação em duas, por motivos de modularidade, ou seja, ao invés de estender cada linguagem de programação usada num determinado computador com primitivas de alto nível, estas são reunidas numa linguagem de controle que é comum a todas. Esta mesma tendência está sendo observada no contexto de ambientes de programação distribuída, embora não se tenha chegado ainda ao mesmo grau de padronização em sistemas distribuídos como nos sistemas centralizados. Uma exceção clara é o caso de interligação de sistemas autônomos, onde haverá de incluir funções de larga escala (criação e ligação de portas, por exemplo) em primitivas da LPP. As características da programação em larga escala serão abordadas mais aprofundadamente no Capítulo V.

II.6 REQUISITOS DE LINGUAGENS PARA PROGRAMAÇÃO DE SISTEMAS DISTRIBUÍDOS

Na seção anterior já foi mostrada a grande influência que

exerce o projeto de desenvolvimento de software sobre as linguagens de programação, que são as ferramentas utilizadas para a sua implementação. Os requisitos de linguagens não são os mesmos para os diferentes tipos de sistemas distribuídos apresentados, e dependendo do sistema haverá requisitos específicos, além de alguns comuns a todos. Nosso propósito aqui é fazer um levantamento global dos requisitos de linguagens para programação de sistemas distribuídos e de aplicações em tempo real, comentando quando for necessário, os requisitos específicos a determinados casos.

Outro ponto que desejamos esclarecer é em relação à linguagem de programação considerada. Os requisitos que serão apresentados a seguir se referem ao que chamamos **linguagem de sistema** (LS) que inclui as características das linguagens analisadas na seção anterior, isto é a LPP e a LPL, independente do modelo adotado, ou seja de serem duas linguagens separadas ou não. Para ilustrar melhor o conceito de linguagem de sistemas vamos apresentar primeiro o modelo de **máquina abstrata de linguagem de sistema** definido por TISATO e ZICARI em [168]. Consideremos um sistema distribuído como um conjunto de nós ligados através de alguma rede de comunicação. Podemos definir vários tipos de nós.

Um **nó físico** é constituído por um ou mais processadores, com memória compartilhada. Ele contém um software básico mínimo escrito numa linguagem de programação usualmente chamada **linguagem de máquina** (LM).

Um **nó lógico** é constituído por um conjunto de processos sequenciais que compartilham objetos. Um nó lógico não pode ser dividido entre vários nós físicos, mas diferentes nós lógicos podem ser associados a um mesmo nó físico. A comunicação entre processos de um mesmo nó físico é realizada através de troca de mensagens ou compartilhamento de memória enquanto entre processos de nós diferentes só é realizada através de troca de mensagens.

Um **nó de tempo real** é um nó lógico para o qual podem ser verificados predicados de tempo real. Neste tipo de nó deve ser definido um tempo único dentro do nó e este deve ter um comportamento determinístico. Somente um nó de tempo real pode ser associado a um nó lógico.

Para dar suporte a uma linguagem de sistemas é necessário prover um nível básico chamado de linguagem de máquina (LM). Neste nível o sistema é visto como um **conjunto de nós físicos independentes**. Para cada nó físico a LM permite descrever as características do hardware mas sem ter uma visão da distribuição. A LM é utilizada para construir o nível básico de gerenciamento de software dos recursos físicos, para cada nó separadamente. Ela deve permitir a implementação do suporte em tempo de execução da LS, deve fornecer portabilidade numa rede de processadores não homogênea e deve ser completamente determinística, isto é, ela deve permitir um controle explícito dos problemas dependentes do tempo fornecendo mecanismos adequados e não políticas embutidas.

Por cima do nível da LM, precisamos enxergar o sistema, no nível da LS, como um **conjunto de nós lógicos ligados**. Para isto a LS deve permitir escrever programas em termos de nós lógicos, fornecendo formalismos linguísticos para comunicação entre nós e primitivas de configuração para mapeamento dos nós lógicos nos nós físicos. A LS será utilizada no nível do sistema operacional e, em determinados casos, no nível da aplicação, segundo a caracterização colocada no início do capítulo, precisando ou não de algumas extensões, mascarando certos aspectos físicos da arquitetura.

Podemos encontrar algumas discussões na literatura [59, 114, 168, 81, 9] sobre este tema levantando diferentes requisitos que procuraremos organizar por tópicos, considerados os mais relevantes.

II.6.1 TIPOS E FACILIDADES DE ABSTRAÇÃO

Definição e verificação de tipos são conceitos fundamentais das linguagens modernas de programação. A verificação forte de tipos aumenta a confiabilidade e a legibilidade dos programas. A linguagem deve poder implementar e manipular tipos abstratos de dados, tais como monitores, módulos, pacotes, etc. Em particular, recursos físicos presentes nos vários nós da rede devem aparecer como instâncias de tipos abstratos de dados predeclarados pelo sistema, de maneira que as operações correspondentes possam ser uniformizadas, e que seja possível uma verificação estática. Por

exemplo, se considerarmos um tipo abstrato representado por um módulo que exporta as operações permitidas, estas estarão especificadas na interface do módulo escrito na LPP. A especificação da interface permite que cada módulo seja compilado separadamente dos outros e que seja completamente verificado em relação à sua interface. A LPL é responsável pela verificação de tipos da interface de cada módulo em relação aos recursos providos pelos outros módulos e especificados nas suas próprias interfaces. LISKOV afirma em [94] que embora a linguagem de sistema deva permitir uma verificação forte de tipos para aumentar a eficiência, é necessário poder fazer conversões de tipo, para considerar diferentes enfoques do mesmo objeto, e para permitir conversões de representação interna de dados para externa e vice-versa. Isto é particularmente imprescindível no caso de serem ligadas máquinas diferentes ou quando são utilizadas linguagens do tipo LPP distintas. Nestes casos é necessário fazer a tradução da representação de uma máquina para a da outra, e também quando é utilizada uma linguagem intermediária com representações diferentes daquelas das máquinas envolvidas. Segundo APPELBE [9], a eficácia da verificação de tipos está ligada com o grau de conversão automática de tipo permitido pela linguagem.

II.6.2 COMPILAÇÃO SEPARADA

É necessário que um módulo possa ser compilado sem conhecer o código dos outros módulos que compõem o resto do programa. As compilações separadas devem preservar a verificação forte de tipos. Para isto, é necessário que as interfaces dos módulos sejam declaradas explicitamente como foi visto no ponto anterior, e que seja feito um desenvolvimento ascendente ("bottom-up"), isto é, os módulos que implementam abstrações subsidiárias devem ser compilados antes dos módulos que usam tais abstrações. No caso de linguagens que separam a parte de especificação de uma unidade de compilação, da parte de implementação (como por exemplo, ADA, Modula-2, MESA), só será necessário ter compilado as partes de especificação das unidades cujas interfaces são importadas pela unidade a ser compilada. Geralmente a informação das seções de especificação é armazenada numa base de dados que

será utilizada durante a compilação pelas outras especificações. A compilação das seções de código correspondentes pode então ser feita em qualquer momento e em qualquer ordem.

Uma das vantagens principais da compilação em separado é que as unidades de compilação podem ser armazenadas numa biblioteca, prontas para serem reusadas posteriormente para montar programas diferentes. Este requisito, muito importante para o desenvolvimento de sistemas grandes de software, é ainda necessário para fazer configuração dinâmica de um sistema, por exemplo, substituindo um módulo por outro com diferente implementação e mantendo a mesma interface.

II.6.3 MODULARIDADE

O conceito de módulo é fundamental para a definição da linguagem. Ele pode ser visto como uma cerca de proteção ao redor de uma coleção de processos, procedimentos e dados. O módulo pode alterar o escopo normal de identificadores, restringindo o acesso a seu conteúdo de uma forma bem definida. Pode prover também exclusão mútua no acesso a dados compartilhados, criando uma forma de abstração de dados que pode ser explorada pelo programador.

O módulo, como foi colocado por TISATO e ZICARI em [168], é:

- o mecanismo básico para suportar abstrações;
- a unidade de compilação separada;
- o mecanismo básico para garantir proteção;
- a unidade para configuração do sistema;
- a unidade para alocação física;
- o suporte para a divisão em camadas do sistema.

Como já foi mencionado, deve ser possível definir diferentes instâncias de um módulo. Esta característica é essencial para poder construir um sistema a partir de uma biblioteca de módulos contendo várias versões de módulos reutilizáveis.

II.6.4 CONCORRÊNCIA

A linguagem deve permitir a definição de atividades

concorrentes, isto é, um conjunto de processos que tem o potencial de executar em paralelo. Para isto é necessário ter mecanismos adequados de comunicação e sincronização entre processos, para transmitir dados de um para o outro e em determinados casos, para forçar uma sequência de execução específica. Diferentes operações primitivas são apresentadas nas linguagens paralelas.

Para a sincronização algumas variáveis sem um valor associado, como por exemplo sinais (HOLT (CSP/k [68]), WIRTH (Modula [176]), BRINCH HANSEN (Pascal Concorrente [28]), etc), são explicitamente enviadas e esperadas. São utilizadas também expressões condicionais para sincronizar a execução de regiões críticas e de chamadas de procedimentos (WIRTH (EDISON [30]), BRINCH HANSEN (DP [29], Pascal Concorrente [28]), ICHBIAH (ADA [73]), ANDREWS (SR [4]), etc). A sincronização é implícita na semântica de chamadas de co-rotinas ou procedimentos (WIRTH (Modula-2 [180]), BRINCH HANSEN (DP [29], Pascal Concorrente [28], EDISON [30]), ICHBIAH (ADA [73]), ANDREWS (SR [4]), etc) e na invocação de processos (BRINCH HANSEN (DP [29], Pascal Concorrente [28]), HOLT (CSP/k [68]), ANDREWS (SR [4]), COOK (*MOD [42]), ICHBIAH (ADA [73]), MITCHELL (MESA [113]), etc). Em contraposição às técnicas de sincronização implícita, uma linguagem pode oferecer expressões de sincronização explícita, separadas do código executável, tais como expressões de trajeto ("path expressions") (Path Pascal [34]).

Os mecanismos de comunicação são também variados. No caso do sistema possuir alguma memória comum, as variáveis compartilhadas são protegidas por um monitor ou alguma outra construção projetada para prover exclusão mútua no acesso aos dados. Os processos se comunicam indiretamente por chamadas de procedimentos do monitor. Mas no caso de não existir memória compartilhada é mais utilizado a comunicação direta entre processos através da troca de mensagens. A partir da técnica básica foram criados vários tipos de primitivas de troca de mensagens com diferentes níveis de estruturação ,que serão analisados no próximo capítulo. Em particular o conceito de porta é outra variante, muito utilizada em sistemas distribuídos: o processo pode ter acesso a uma porta para enviar ou receber mensagens ou para ambas as operações (portas de entrada/saída),

estabelecendo uma rede de comunicação que, dependendo do tipo de ligações utilizadas entre as portas, pode ser dinâmica.

Vemos que existem diferentes níveis de mecanismos de comunicação e sincronização. Quanto mais alto o nível dos mecanismos que a linguagem oferece, menos complexo será o programa, mas o escalonamento dos processos estará restringido por esses mecanismos. Um enfoque alternativo é adotado por Modula-2 que provê um mecanismo de baixo nível, a co-rotina, que pode ser utilizada para implementar diferentes políticas de comunicação entre processos de mais alto nível. Estas políticas são definidas para uma determinada aplicação, ou podem fazer parte de uma biblioteca disponível ao usuário da linguagem, como já foi mencionado anteriormente.

Outro ponto a ser levantado é o da criação de processos concorrentes. Existem duas alternativas, a criação estática no início da execução do sistema, sendo que o número, o tipo e a identificação de cada processo é conhecida em tempo de compilação, e a criação dinâmica, na medida que for necessário criar processos novos, durante a execução do sistema. Os processos estáticos existem enquanto o sistema estiver executando, mesmo depois deles não terem mais razão de ser, enquanto os processos dinâmicos geralmente desaparecem depois de terminados. A criação dinâmica pode ser feita de duas maneiras. Um processo pode ser definido no programa e depois serem criadas uma ou várias instâncias dele durante a execução. A outra forma de criação dinâmica de processos ocorre em linguagens nas quais os processos podem ser tipos e variáveis com valores que podem ser criados em tempo de execução. É importante, finalmente distinguir entre a criação estática/dinâmica de processos, e o conceito relacionado de configuração estática/dinâmica, que será tratada no Capítulo V.

II.6.5 CONFIGURAÇÃO

A linguagem deve permitir definir: os componentes do sistema e as suas interligações, de que forma alguns módulos mais complexos e o sistema como um todo são construidos a partir de módulos já existentes, ou seja a arquitetura lógica do sistema, e por último o mapeamento dos nós lógicos nos nós físicos. Estas

funções podem ser feitas por um subconjunto da linguagem, chamado de LPL, que como, já foi visto anteriormente, pode ser considerado como uma linguagem separada, ou, pelo contrário, como uma extensão da LPP. É desejável tratar os módulos como "schemata" de forma que, provendo valores para determinados parâmetros, possam ser criados módulos específicos. Para isto a LPL deve possuir primitivas básicas para criação, destruição e ligação de instâncias de módulos. É desejável também que a LPL contenha, além das primitivas para especificar a configuração do sistema, mecanismos para controlar versões e manter bibliotecas de definições e implementações de módulos. Estes módulos evoluem à medida que os sistemas são desenvolvidos, e a LPL deve possibilitar a implementação das mudanças que a evolução provoque.

As formas de definir interfaces podem ser caracterizadas fundamentalmente de duas maneiras: as interfaces de procedimentos e as portas. Estas correspondem respectivamente às duas formas diferentes de comunicação e sincronização entre processos: a chamada (remota ou não) de procedimentos e a troca de mensagens. No primeiro caso a ligação entre módulos é feita através da associação do conjunto de procedimentos exportados por um módulo com um ou mais procedimentos importados pelo outro, que logicamente têm que coincidir nos tipos. No caso de uso de portas existem vários esquemas diferentes com portas de entrada, portas de saída ou portas de entrada e saída, cada uma tendo algum tipo de mensagem associada. As ligações entre módulos são realizadas através da ligação das portas exportadas por um módulo e importadas pelo outro tendo que coincidir como no caso anterior, os tipos das portas. Estas duas formas de definir interfaces serão discutidas detalhadamente nos capítulos seguintes.

II.6.6 POLÍTICAS DE GERENCIAMENTO

A definição da linguagem de sistema não deve conter mecanismos de gerenciamento embutidos. Deve ser possível definir políticas adequadas para o gerenciamento de recursos (processador, memória, periféricos, etc.) de acordo com os requisitos específicos de cada aplicação. Em particular para implementar sistemas em tempo-real é necessário definir

explicitamente as políticas de gerenciamento dos processos ligados a determinados dispositivos de hardware que estão sendo controlados. Seria difícil definir mecanismos de gerenciamento gerais sem conhecer em certo nível a distribuição real do sistema, e, mais ainda, que estes mecanismos sejam adequados para diferentes tipos de aplicações. Esta restrição é necessária também para que o suporte em tempo de execução da linguagem de sistema seja simples. É então fundamental que a linguagem de sistema dê acesso a mecanismos que permitam programar as políticas para escalonadores de processadores, gerenciadores de memória, gerenciadores de recursos, etc. A modularidade, já discutida, permite que estas políticas de gerenciamento fiquem restritas a módulos específicos, isolados do resto do programa, escondendo detalhes de nível mais baixo, e facilitando também a substituição de uma política por outra quando for necessário.

II.6.7 EXECUÇÃO NÃO DETERMINÍSTICA

Em várias aplicações de sistemas distribuídos, é importante poder expressar um certo não determinismo na execução do sistema, isto é, que, para várias execuções sobre os mesmos dados de entrada, se possa obter conjuntos diferentes de dados de saída. Algumas linguagens possuem declarações especiais para expressar explicitamente o não determinismo, (por exemplo regiões ou comandos com guarda ("guarded commands")) (BRINCH HANSEN (DP [29]), HOARE ("Communicating Sequential Processes" [68]), (ICHBIAH (ADA [73]), ANDREWS (SR [4]), etc). Enquanto em outras o não determinismo acontece através de algum escalonador de processos ou de algum mecanismo de tratamento de filas, os quais são inacessíveis ao programador. Esta característica não é adequada no caso de sistemas em tempo real como será visto a seguir.

II.6.8 SUPORTE DE TEMPO REAL

Para aplicações em tempo real a linguagem deve possuir características que permitam ao programador controlar explicitamente a ordem de execução dos processos, fazer temporização ("time-out") e utilizar gerenciadores de exceções na

fase de comunicação entre processos, comunicar-se diretamente com dispositivos de hardware existentes no ambiente, etc. Tais características tornam a programação de aplicações em tempo real mais fácil e confiável e satisfazem o requisito fundamental de comportamento determinístico.

II.6.9 TRATAMENTO DE EXCEÇÕES

A linguagem deve possuir facilidades para que o programador defina ações a serem tomadas quando ocorrerem condições excepcionais durante o tempo de execução a fim de evitar que o sistema como um todo tenha um comportamento errado ou precise até ter sua execução discontinuada. É necessário poder isolar as condições excepcionais, e adotar soluções locais que permitam contornar o problema ocorrido, para que o sistema continue a sua execução. Já foram citadas anteriormente algumas linguagens que oferecem tratamento de exceções (ADA (ICHBIAH [73]), MESA (MITCHELL et alii [113]), CLU (LISKOV et alii [95])).

II.6.10 SUPORTE PARA VERIFICAÇÃO DE PROGRAMA

A lingugem deve possuir características que ajudem os programadores ou verificadores automáticos a determinar a correção lógica do programa. Essas características podem estar relacionadas com construções sintáticas, incluindo especificação formal de processos e suportes semânticos como, por exemplo, convenções de chamada de processos e de monitores que restringem as instâncias de comunicação a estruturas hierárquicas não recursivas.

Foram apresentados aqui alguns dos requisitos considerados mais importantes para a linguagem de sistema de programação de sistemas distribuídos. Como já foi mencionado, alguns requisitos são mais adequados para determinados tipos de sistemas distribuídos enquanto outros são gerais. Estes requisitos nortearam o trabalho desenvolvido nesta tese em relação à definição do ambiente de programação baseado na linguagem Modula-2 que já possuia vários dos requisitos aqui levantados.

CAPÍTULO III

PROGRAMAÇÃO EM PEQUENA ESCALA: MECANISMOS DE COMUNICAÇÃO E SINCRONIZAÇÃO

Como já foi exposto no capítulo anterior, para a estruturação modular de programas distribuídos, podemos diferenciar duas atividades, distintas: programação em pequena escala e programação em larga escala. Concentraremo-nos aqui na primeira atividade analisando os mecanismos de comunicação e sincronização utilizados pela maioria das linguagens que encontramos na literatura, para programar módulos individuais.

Em sistemas concorrentes, baseados na utilização de processos paralelos que se comunicam e se sincronizam, a escolha dos mecanismos de comunicação e sincronização de processos tem importância fundamental. Para que essa escolha seja bem feita, é necessário que se tenha um bom conhecimento das características que se quer projetar, bem como dos mecanismos existentes, utilizados para realizar comunicação e sincronização entre processos.

Uma grande variedade de propostas desses mecanismos pode ser encontrada na literatura (ANDREWS e SCHNEIDER [5], BRINCH-HANSEN [26], LISTER [101]), destacando-se as seguintes: semáforos, eventos, regiões críticas simples e condicionais, monitores, expressões de trajeto, etc. O conceito de monitor (HOARE [67]), em particular, apareceu fortemente influenciado pelo conceito de programação estruturada, e foi logo incorporado em algumas linguagens de alto nível para programação concorrente, tais como: Pascal Concorrente (BRINCH HANSEN [28]), Modula (WIRTH [176]), CSP/k (HOLT et alii [70]), MESA (MITCHELL et alii [113]), entre outras. Todos esses mecanismos citados baseiam-se no acesso disciplinado a um ambiente global, acessível aos processos concorrentes fortemente acoplados e podendo compartilhar memória (ver seção II.2).

Devido ao fato de não sempre existir memória compartilhada nos sistemas distribuídos, os mecanismos de comunicação e sincronização, anteriormente citados, apresentam-se como inadequados para utilização nesses sistemas. A comunicação e sincronização de processos, localizados em estações diferentes,

passa agora a ser realizado por meio de troca explícita de mensagens através da rede de comunicação.

Visando adquirir elementos para a definição de um ambiente de programação para projetar sistemas distribuídos, optamos por fazer um levantamento dos mecanismos de comunicação e sincronização baseados no uso de memórias distribuídas, já publicados. Ao mesmo tempo procuramos, também, realizar uma análise das características marcantes desses mecanismos. Cabe esclarecer que serão tratados os mecanismos de diferentes graus de complexidade, desde o mais simples, que consiste da troca explícita de mensagens, até os mais estruturados, que são implementados em termos de mecanismos mais básicos. Entre os mecanismos mais estruturados destacam-se o "rendezvous" e a chamada remota de procedimento, que permitem estender o conceito de monitor. Quando os monitores estão localizados em endereços distintos dos programas que os chamam, é necessário implementar mecanismos, que permitam que uma máquina (ou máquina virtual) chame outra, para pedir a execução de um determinado procedimento pertencente ao monitor localizado nesta segunda máquina.

Como exemplo, podemos citar algumas linguagens que oferecem mecanismos de troca de mensagens assíncronas, tais como Gypsy (GOOD et alii [61]), Plits (FELDMAN [55]), *MOD (COOK [42]), Pascal M (ABRAMSKY e BORNAT [1]), a proposta de LISKOV [94] e Menyma (KOCH e MAINBAUM [84]). Entre as linguagens que fornecem mecanismos síncronos simples, como troca de mensagens síncronas, é "Communicating Sequential Processes" (CSP, HOARE [68]), e linguagens com mecanismos mais estruturados como chamada remota de procedimento e "rendezvous" incluem DP (BRINCH HANSEN [29]) e ADA (ICHBIAH et alii [73]), respectivamente.

Será analisado também o conceito de porta que foi introduzido como forma de implementar uma nomeação indireta na troca de mensagens. Com a evolução deste conceito, foi-lhe associado um tipo de dado identificando as mensagens que podem ser transmitidas através da porta. Outras características serão apresentadas, tais como as funções que as portas exercem, as ligações necessárias para implementar diferentes topologias, e a contribuição dada para o conceito de modularidade.

Para concluir é abordada a discussão sobre a dualidade dos dois estilos de programação em pequena escala, definidos por

LAUER e NEEDHAM [90], e será estendida a sua discussão para sistemas distribuídos.

III.1 MECANISMOS DE COMUNICAÇÃO E SINCRONIZAÇÃO

Os mecanismos mais simples são aqueles que utilizam primitivas básicas como **envia** e **recebe**, dispostas convenientemente nos processos de maneira a permitirem sincronização e comunicação.

A primitiva **envia** transfere informação do processo origem (emissor) para o processo destino (receptor), enquanto a primitiva **recebe**, por sua vez, como seu nome indica, permite que o receptor receba a referida informação enviada pelo emissor. O comportamento dos processos emissor e receptor, ao executarem as respectivas primitivas, dependerá do tipo de sincronização e endereçamento adotados.

A sincronização refere-se à restrição no andamento dos processos. Uma primitiva é **bloqueada (síncrona)** quando o processo que a executa fica bloqueado até que a operação associada tenha completado a sua execução. Uma primitiva é **não bloqueada (assíncrona)** quando o processo que a executa continua seu andamento tão logo que ela seja concluída localmente.

As primitivas básicas **envia** e **recebe** podem apresentar-se de forma bloqueada ou não bloqueada. Como foi colocado por MANNING et alii [107], existem quatro modos de sincronização e comunicação entre processos, dependendo das combinações possíveis de primitivas bloqueadas e não bloqueadas, que são as seguintes:

i) Envia Bloqueado - Recebe Bloqueado (Eb-Rb)
ii) Envia Bloqueado - Recebe Não Bloqueado (Eb-Rnb)
iii) Envia Não Bloqueado - Recebe Bloqueado (Enb-Rb)
iv) Envia Não Bloqueado - Recebe Não Bloqueado (Enb-Rnb)

O modo (Eb-Rb) é chamado totalmente síncrono, enquanto o modo (Emb-Rnb) é totalmente assíncrono. Os outros dois modos restantes são denominados semi-síncronos. Estas quatro combinações de primitivas de emissão e recepção constituem mecanismos simples de comunicação e sincronização.

III.1.1 MECANISMOS ASSÍNCRONOS

São considerados mecanismos de comunicação e sincronização assíncronos as combinações de primitivas **envia** e **recebe** nos casos semi-síncronos e totalmente assíncronos. O assincronismo será assegurado pela presença de pelo menos uma primitiva **envia** ou **recebe** não bloqueada.

Tanto na emissão quanto na recepção assíncronas, o processo, que executar a primitiva, não ficará bloqueado esperando a sincronização com o outro processo.

No caso da emissão assíncrona, o processo continuará a sua execução, logo depois da sua mensagem ser aceita para transmissão, sem precisar esperar por nenhuma resposta.

No caso da recepção assíncrona, o processo deverá continuar a sua execução, mesmo quando não existir nenhuma mensagem pronta para ser recebida. Dependendo da implementação, a mensagem poderá, ou não, ser ignorada pelo processo receptor quando chegar atrasada. Neste último caso deverá existir alguma primitiva síncrona no receptor a ser executada posteriormente para conferir a sua chegada.

A principal vantagem deste tipo de primitivas é que elas permitem explorar ao máximo o paralelismo de um conjunto de processos concorrentes, em contraposição ao que acontece com as primitivas síncronas. Em particular, o modo totalmente assíncrono é aquele que oferece a maior possibilidade de paralelismo. Porém, geralmente deverão ser acrescentadas algumas primitivas para sincronização entre os processos concorrentes, já que eles nunca serão totalmente independentes.

III.1.1.1 EMISSÃO ASSÍNCRONA

LISKOV [28] defende o uso da primitiva de emissão não bloqueada, por considerá-la mais flexível e de mais baixo nível que a primitiva de emissão bloqueada, uma vez que, com a primeira, é possível implementar os outros tipos de protocolos de comunicação, em particular, a comunicação síncrona, embora a inversa não seja verdadeira. Por exemplo, a primitiva de emissão

assíncrona permite implementar naturalmente a emissão de mensagens do tipo difusão ("broadcasting"). Este tipo de mensagem é muito utilizado em sistemas de controle, e tem como objetivo transmitir alguma informação relevante para todos os processos do sistema, sem necessidade de resposta associada.

Embora a utilização de primitivas de emissão assíncrona em conjunto com recepção síncrona, conforme a proposta de LISKOV [28] e a linguagem Menyma (KOCH [83]), forneça uma sincronização adequada, não se consegue controle de fluxo, já que o emissor pode continuar a execução de forma independente do receptor. Este comportamento implica na formação de filas potencialmente ilimitadas de mensagens que precisam ser armazenadas. As filas são geralmente implementadas ao nível do sistema de comunicação, e o seu gerenciamento é a dificuldade mais séria que estas primitivas apresentam. O esgotamento de memória para armazenar as mensagens constitui-se num problema, quando o sistema de comunicação não possui capacidade de requisitar mais memória, pelo fato de estar num nível inferior ao do gerenciamento de memória virtual. O método de gerenciamento de memória para armazenamento de mensagens pode fazer com que a memória disponível para um processo seja afetada pelo comportamento dos outros processos, podendo-se até chegar a uma situação de bloqueio perpétuo.

Uma das soluções, propostas para o problema de tratamento de filas, é limitar o tamanho das filas de mensagens de cada processo. Se o limite de tamanho das filas for explícito, isto é, se seu controle for da responsabilidade do programador, tanto a compreensão do programa, quanto a verificação de sua correção e a análise de seu desempenho, poderão tornar-se mais complicadas. Se esse limite for implícito, ou seja, se o tamanho das filas for controlado pelo núcleo, haverá certamente perda de velocidade e eficiência e poderão até ocorrer problemas de bloqueio perpétuo, caso não haja políticas adequadas para evitá-los. Isto poderá provocar um atraso não desprezível na transferência de mensagens entre processos, contrariamente ao desejado.

O tratamento de filas de mensagens introduz, também, o problema decorrente da impossibilidade de se poder descrever o estado total do sistema através do estado dos processos. Para analisar a integridade do sistema devem-se incluir os estados das

áreas de mensagens, e considerar a condição de área esgotada.

Dependendo das aplicações a serem desenvolvidas, existem algumas soluções para o tratamento de filas de mensagens. Por exemplo, no artigo de KRAMER et alii [85] é apresentado o caso de um sistema de controle em tempo real, no qual as mensagens enviadas são leituras de sensores, alarmes, etc. Argumenta-se que, neste caso, o que interessa é sempre a última mensagem enviada, e não a sua história. Portanto, é suficiente manter uma área de mensagens com capacidade unitária, de forma que ali esteja armazenada a última mensagem enviada, mesmo que a anterior não tenha sido recebida. Nas versões mais recentes do projeto CONIC [86] este tratamento foi estendido implementando uma área de mensagens de tamanho **N** sendo que são sempre armazenadas somente as últimas **N** mensagens.

Outra forma de tratamento de mensagens na comunicação assíncrona é através do uso de nomes globais, geralmente chamados de **caixas postais**, associados a áreas de armazenamento de mensagens. A caixa postal pode aparecer como o nome destino de uma primitiva de envio ou como o nome fonte de uma primitiva de recepção, executadas por processos. As mensagens enviadas para uma determinada caixa postal podem ser recebidas por qualquer processo que execute uma primitiva de recepção nomeando essa caixa postal.

Este esquema é particularmente apropriado no caso de interações de tipo cliente/servidor. Os processos clientes mandam seus pedidos de serviço a uma determinada caixa postal e os processos servidores recebem os pedidos de serviço a partir dessa caixa postal. Esta implementação porém, pode ser custosa, como foi colocado por GELERNTER e BERNSTEIN em [57], sem uma rede de comunicação especializada (no caso de ter vários servidores equivalentes). Quando uma mensagem é enviada para a caixa postal, ela deve ser transmitida para todos os lugares nos quais pode ser executada uma operação de recepção sobre essa caixa postal, isto é, para todos os servidores equivalentes existentes. Depois da mensagem ter sido recebida por um deles, é necessário avisar aos outros que a mensagem não está mais disponível.

No caso particular de ter vários clientes e um único servidor, a caixa postal é associada ao servidor, que será seu proprietário. A implementação é muito mais simples já que todas

as operações de recepção sobre essa caixa postal serão executadas pelo único servidor e não acontecerão os problemas levantados anteriormente. Serão analisadas mais adiante com detalhe, as diferentes formas de endereçamento das primitivas de comunicação e sincronização.

III.1.1.2 RECEPÇÃO ASSÍNCRONA

A primitiva de recepção assíncrona, mesmo sendo pouco usada, é apropriada para alguns casos particulares.

Por exemplo, no caso do sistema prever aparecimento de mensagens imprevisíveis como acontece com os alarmes, é razoável que os processos, com capacidade de recebê-los, contenham várias primitivas de recepção não bloqueadas espalhadas. As posições dessas primitivas, ao longo do processo, deverão satisfazer algumas restrições, dentre elas a restrição de tempo, uma vez que o ponto de ocorrência de alarme é imprevisível.

Outro exemplo de uso de recepção não bloqueada é o caso de comunicação entre processos, no qual um processo envia várias mensagens para outro, podendo receber uma mensagem de volta em qualquer instante. Este exemplo está ilustrado por MANNING et alii [107], através da implementação de transmissão de pacotes de N mensagens de tamanho fixo. O emissor deve executar N-1 emissões não bloqueadas, cada uma seguida de uma recepção não bloqueada. Se nenhuma mensagem de retorno chegar antes da emissão N, o processo emissor deverá executar uma recepção bloqueada e esperar a resposta.

III.1.1.3 OUTRAS PRIMITIVAS ASSÍNCRONAS

Outro tipo de primitivas assíncronas é o das **primitivas condicionais**, apresentado em GENTLEMAN [58]. O processo, que executar a emissão condicional, enviará a mensagem somente se o processo receptor estiver bloqueado esperando por ela, caso contrário retornará uma indicação de falha. Da mesma forma, o processo que executar a recepção condicional só receberá a mensagem se ela estiver pronta, caso contrário retornará uma indicação de falha. Nos dois casos, o processo que não conseguir

executar a primitiva não ficará bloqueado. Estas primitivas são utilizadas para evitar bloqueio, na espera de eventos menos críticos, numa situação onde um processo precise coordenar vários tipos de eventos. A desvantagem deste esquema é que o processo deve testar todos esses eventos menos críticos e ter disponível algum outro tipo de mecanismo de sincronização.

Existe ainda outro tipo de primitiva assíncrona apresentada por GENTLEMAN [58], chamada **resposta assíncrona** ("reply"), utilizada no sistema operacional THOTH [37]. Esta primitiva deve ser usada em conjunção com primitivas de emissão e recepção síncronas. Neste caso, a primitiva de emissão síncrona deve bloquear o processo emissor até que este receba uma resposta emitida pelo processo receptor. A primitiva assíncrona de resposta especial, situada no processo receptor, atua de forma a desbloquear o processo emissor, sem interferir no andamento do processo receptor. De acordo com GENTLEMAN [58], o uso apropriado das duas primitivas bloqueadas, em conjunto com a resposta não bloqueada, evita a necessidade de outras primitivas não bloqueadas, e de mecanismos de sincronização adicionais.

III1.1.4 ALGUMAS CONSIDERAÇÕES SOBRE AS POSSÍVEIS APLICAÇÕES

Os mecanismos de comunicação assíncronos podem ser utilizados em diferentes tipos de aplicações, como foi colocado por KIRNER em [81]. A combinação Eb-Rnb é adequada para sistemas com estrutura do tipo cliente/servidor, onde cada cliente envia pedidos de execução de operações do servidor, enquanto o servidor procura atender a todos os clientes num sistema de varredura. Neste caso a comunicação deve ser considerada bidirecional já que o emissor fica bloqueado esperando alguma resposta do servidor, que lhe será enviada depois de executar a operação requisitada, através de outra primitiva.

A combinação Enb-Rb pode ser usada em sistemas com estrutura composta de processos cooperativos, no caso de um processo querer enviar uma informação de interesse exclusivo do outro processo, de maneira a poder continuar a sua execução sem necessidade de sincronização com o outro processo naquele ponto.

Em sistemas onde vários processos cooperam aleatoriamente com um único processo é apropriado utilizar a combinação

totalmente assíncrona Enb-Rnb. Neste caso, os emissores não precisarão ficar bloqueados, havendo somente a necessidade do receptor fazer uma varredura de recepção sem bloquear-se. Em [107] é chamada a atenção em relação a um problema decorrente do uso desta combinação. Através da programação de alguns exemplos é mostrado claramente como, além de ser possível o envio de várias mensagens, as execuções de primitivas de recepção não bloqueadas podem tratar várias mensagens de forma intercalada. Nesse caso, deve-se tomar o cuidado de incluir algum mecanismo de exclusão mútua para garantir a indivisibilidade da mensagem.

III.1.2 MECANISMOS SÍNCRONOS

Estes mecanismos provêem uma forma simples e elegante de combinar a função de sincronização com a transferência de dados, não sendo necessária nenhuma primitiva independente de sincronização. Eles são caracterizados por serem síncronos e por não precisarem de memória para armazenar as mensagens, já que a transmissão das mensagens só será feita na hora em que os dois processos que se comunicam estiverem sincronizados.

III.1.2.1 DESCRIÇÃO DOS DIFERENTES TIPOS

De acordo com a análise feita por SEGRE e KIRNER [141] podemos diferenciar os mecanismos síncronos em três níveis de estruturação:

- emissão e recepção bloqueadas simples
- "rendezvous"
- chamada de procedimento

A **emissão e recepção bloqueadas simples** consiste de um par de primitivas **envia** e **recebe** bloqueadas colocadas em processos distintos, de forma que o processo que envia a mensagem se bloqueia até que o processo receptor esteja pronto para recebê-la. Da mesma maneira, se não houver nenhuma mensagem já emitida esperando, o processo receptor se bloqueia na operação de recepção até que uma seja enviada. A transmissão da mensagem representa o ponto de sincronização entre o emissor e o receptor

e, depois da sua execução, os dois processos continuam as suas respectivas execuções em paralelo, isto é, o receptor pode utilizar a informação transmitida em paralelo com a execução do emissor. Um exemplo de linguagem que usa este tipo de comunicação é "Communicating Sequential Processes", CSP (HOARE [58]), que contém comandos de entrada e saída para sincronizar a comunicação entre processos paralelos. A comunicação é realizada quando o emissor executa uma instrução de saída e o receptor uma de entrada.

A forma bloqueada da primitiva **recebe** é a mais comum devido ao fato que o processo receptor geralmente não tem nada para fazer enquanto espera a recepção da mensagem. Entretanto, a maioria das linguagens e dos sistemas operacionais fornecem meios de contornar esse bloqueio total, implementando recepção condicional, que permite ampliar a forma de recepção, ou através do uso de comandos guardados ("guarded commands") introduzidos por DIJKSTRA [48].

O **"rendezvous"** caracteriza-se por apresentar uma estrutura de comunicação e sincronização na qual um processo comanda a execução de um trecho de programa, previamente definido, associado à recepção num outro processo. O processo que envia a mensagem fica bloqueado até que o processo receptor envie alguma resposta depois de ter utilizado a informação transmitida, ou seja, depois de tê-la recebido e executado algum código para tratá-la. Analogamente ao caso anterior, o processo receptor fica bloqueado na operação de recepção até receber alguma mensagem. Uma linguagem que usa este tipo de sincronização é ADA (ICHBIAH et alii [73]), que possui o conceito de tarefa que pode ser executada em paralelo com outras tarefas. A comunicação e sincronização são realizadas a partir do momento em que a primeira tarefa, que executa uma chamada para um ponto de entrada de uma segunda tarefa, é sincronizada com a segunda quando esta alcança uma instrução de aceitação da chamada. Uma vez realizada a sincronização, o receptor, depois de receber a informação transmitida, executa o corpo da instrução de aceitação associada á chamada, enquanto o emissor fica bloqueado até o receptor acabar esta execução, quando então o "rendezvous" se completa. Nesse ponto pode ocorrer transferência de parâmetros contendo resultados e, em seguida, as duas tarefas prosseguem em execução

concorrente.

A **chamada de procedimento**, que pode ser local ou remota, representa outro nível de estruturação de mecanismo de comunicação e sincronização síncrono. Este mecanismo pode ser implementado tanto para chamada de procedimentos locais, isto é, para procedimentos localizados fisicamente no mesmo nó no qual foi feita a chamada, ou para procedimentos remotos localizados em nós diferentes.

Dentro de um trecho de programa é executada uma chamada de procedimento geralmente passando parâmetros de entrada e de saída; estes últimos comunicam os resultados da execução do procedimento chamado. Existe então uma transferência do fluxo de controle de execução do trecho que estava executando, antes da chamada de procedimento, para o procedimento chamado. O controle de execução volta para o trecho de programa que fez a chamada somente depois do procedimento ter acabado a sua execução e ter enviado os resultados de volta. Este esquema é parecido com o anterior, com a diferença que neste caso a sua execução será intercalada com a execução da unidade que o contém de forma não determinística, isto é, a recepção não é bloqueada como no caso do "rendezvous".

Este esquema é muito utilizado em sistemas programados em linguagens com interfaces procedimentais. Uma linguagem que usa este tipo de comunicação é "Distributed Processes", DP (BRINCH HANSEN [29]), onde a comunicação entre processos dá-se através de chamadas de procedimentos compartilháveis definidos em outros processos. Por outro lado, a sincronização entre processos ocorre através de comandos não-determinísticos chamados regiões guardadas ("guarded regions") (DIJKSTRA [48]).

Em particular, para o caso de chamadas remotas de procedimentos (RPC), existem implementações que acrescentam este mecanismo para ambientes distribuidos programados em diferentes linguagens. Podemos citar como exemplo a implementação de NELSON e BIRRELL [19] para RPC a ser acrescentada à linguagem MESA de forma que as chamadas de procedimentos fiquem transparentes para o usuário; ele não precisa saber se a chamada é local ou remota. Por outro lado achamos na literatura a implementação de HAMILTON [64] a ser acrescentada à linguagem CLU concorrente, na qual as RPC são explícitas e não transparentes para o usuário. Estas duas

implementações serão analisadas com mais detalhe no próximo capítulo.

Para ser coerentes com as nossas definições do início deste capítulo, precisamos reconhecer que este último mecanismo não se encaixa completamente com a definição de comunicação totalmente síncrona já que a recepção não é totalmente bloqueada. Mas na verdade este mecanismo é apresentado aqui porque ele é considerado como o tipo de nível de estruturação mais alto dentro da comunicação síncrona.

Para ilustrar o que acabamos de colocar podemos citar que tanto o "rendezvous" quanto a chamada de procedimento estão relacionados com a transação do tipo "request-reply" apresentada por KRAMER et alii [85] e por GENTLEMAN [58]. Esse tipo de transação consiste na transferência de duas mensagens: uma de requisição emitida pelo processo requisitante para o processo de serviço, e outra de resposta emitida pelo processo de serviço para o processo requisitante. Com a utilização de primitivas adequadas, esse tipo de transação poderá ser transformado em "rendezvous" ou em chamada de procedimento, desde que, neste último caso, a requisição seja endereçada a um procedimento.

III.1.2.2 COMPARAÇÃO DOS DIFERENTES TIPOS

Dos três tipos de primitivas síncronas, o primeiro, que consiste de emissão e recepção bloqueadas, é o mais simples e de nível mais baixo, o que corresponde a dizer que os outros dois tipos podem ser implementados, usando as primitivas do primeiro tipo.

Estas primitivas, por serem muito simples, têm algumas vantagens importantes. O assincronismo dos processos, estando concentrado no intercâmbio de informação não determinístico, permite que cada processo possa ser considerado como completamente sequencial. Neste esquema não existe transferência de controle entre processos, o que facilita a verificação da correção do sistema, considerado como um conjunto de processos disjuntos.

Este esquema é adequado para comunicação entre processos que precisam de sincronização explícita em diferentes pontos. Sistemas baseados em emissão e recepção bloqueadas usam um

conjunto mínimo de elementos, consistindo de processos e mensagens. Isto permite que, através de uma interface simples de comunicação, sejam realizadas combinações de sistemas.

Os outros dois tipos, "rendezvous" e chamada de procedimento, são mecanismos de comunicação e sincronização de nível mais alto, e, portanto, permitem uma maior estruturação do sistema a ser programado. Para o programador de sistemas esses dois tipos são ferramentas mais poderosas e mais seguras do que as operações de emissão e recepção. Ao nível da implementação para sistemas distribuídos, os dois tipos precisarão utilizar as operações de nível mais baixo, isto é, as chamadas de procedimentos e a passagem de parâmetros serão provavelmente implementadas através de operações de emissão e recepção.

É interessante salientar que, tanto no esquema de emissão e recepção bloqueadas quanto no "rendezvous", o processo receptor tem controle explícito sobre a aceitação das mensagens a serem recebidas, e sobre as operações a serem executadas. Isto não acontece no terceiro esquema, onde as execuções dos procedimentos são feitas de forma não determinística como já foi colocado anteriormente. Porém, pode ser acrescentado não determinismo nos dois primeiros esquemas através do uso de comandos guardados como é feito em CSP e ADA.

Nos últimos dois esquemas, a troca de mensagens é feita através de parâmetros passados nas chamadas das operações.

Uma das críticas colocadas por MANNING et alii [107], em relação ao uso desses dois esquemas mais estruturados, é que existe uma transferência de controle do processo emissor para o processo receptor, e que este precisa necessariamente mandar uma resposta de volta ao emissor para devolver-lhe o controle. No primeiro esquema, mais simples, o problema de transferência de controle não existe, pois o processo emissor não fica bloqueado depois de executar o **envia**, e existe ainda a possibilidade de que o processo receptor, depois de utilizar a mensagem, possa mandar os resultados para qualquer outro processo, sendo ou não o emissor. Por exemplo, a linguagem MCD (RUGGIERO e BRESSAN [137]) oferece na primitiva de envio a opção de indicar para quem precisa ser enviada a resposta.

Outra crítica, em relação aos dois últimos esquemas, é que eles induzem a uma determinada hierarquia de organização dos

processos: o processo chamador está em nível mais alto do que o chamado. O esquema de emissão e recepção bloqueadas é mais geral e flexível e pode ser usado para sistemas hierárquicos ou não. Esta crítica pode ser invalidada em parte, pelas discussões mais atualizadas que serão mencionadas no próximo capítulo, sobre o consenso cada dia maior do uso de mecanismos síncronos para sistemas distribuídos, e em particular sobre o uso de RPC. Os sistemas distribuídos são, na sua maioria, estruturados na forma cliente/servidor e portanto o uso de um mecanismo que induza a uma determinada hierarquia é considerado uma vantagem para a estruturação do projeto do sistema.

III.1.2.3 VANTAGENS E DESVANTAGENS DOS MECANISMOS SÍNCRONOS

Além das vantagens que já foram colocadas anteriormente, as primitivas síncronas apresentam também a vantagem de não esconder nenhum tratamento de armazenamento de mensagens, isto é, não existe gerenciamento de áreas de memória ("buffers") implícito na implementação das primitivas. Desta maneira, evitam-se aqui os problemas decorrentes desse gerenciamento, que aparecem e foram analisados no caso das primitivas assíncronas.

O modo de comunicação totalmente síncrono garante a indivisibilidade das transmissões de mensagens, já que um processo só pode enviar ou receber uma mensagem de cada vez.

A maior desvantagem das primitivas síncronas é a inibição de paralelismo na execução dos processos. Um processo, bloqueado por querer enviar ou receber uma mensagem, não pode continuar executando comandos que poderiam ser independentes da comunicação. Nos esquemas mais estruturados, tais como "rendezvous" e chamada de procedimento, a inibição de paralelismo pode até ser maior, já que o processo emissor fica bloqueado até que seja completada a execução da operação chamada.

Outra dificuldade é que alguns conjuntos de primitivas síncronas não permitem que um processo fique bloqueado, esperando por mais de um tipo de evento: isto faz com que mensagens prontas para serem tratadas, fiquem esperando devido a uma determinada ordem imposta na recepção. Para evitar este problema, a linguagem CSP (HOARE [68]) prevê a colocação de comandos de recepção nos comandos guardados, de forma que um processo possa ficar

bloqueado, esperando por diferentes tipos de mensagens. Analogamente, em ADA (ICHBIAH et alii [73]), existe o comando "select" que permite a um processo ficar bloqueado, esperando a chamada de diferentes entradas.

O uso de primitivas síncronas pode produzir modelos, baseados em processos, pouco adequados para determinados propósitos. Por exemplo, processos que gerenciam recursos através de primitivas síncronas, normalmente executam completamente um pedido de usuário antes de aceitar outro. Isto faz com que o gerenciamento seja pouco eficiente, podendo até transformar-se em gargalo do sistema. Para resolver este problema foram sugeridas algumas soluções como o modelo do proprietário (CHERITON et alii [37]), e o modelo do administrador (GENTLEMAN [58]). Este último consiste num processo que contabiliza requisições de processos usuários e ofertas de alocação de recursos, sem ficar bloqueado, e podendo otimizar o gerenciamento dos recursos.

Em sistemas distribuídos sem memória compartilhada é, às vezes, necessário implementar armazenamento de mensagens. Isto pode ser feito explicitamente, interpondo-se um processo intermediário, a cada par de processos comunicantes, que gerencie áreas de memória, e que ofereça operações para colocar e retirar mensagens. Esta solução, assim como aquela do administrador, aumenta o paralelismo do sistema e permite implementar comunicação parcialmente assíncrona entre processos. Por outro lado, o acréscimo de processos intermediários pode tornar-se muito elevado, fazendo com que a solução deixe de ser apropriada.

Outra desvantagem reside na dificuldade de implementação de emissão de mensagens do tipo difusão ("broadcasting"), que é muito utilizada em sistemas de controle, a partir de primitivas de emissão bloqueada, como já foi mencionado anteriormente.

III.1.3 FORMAS DE IDENTIFICAÇÃO E ENDEREÇAMENTO NOS MECANISMOS DE COMUNICAÇÃO E SINCRONIZAÇÃO

Uma característica importante dos mecanismos de comunicação e sincronização entre processos concorrentes é a forma de identificação, tanto dos processos receptores, quanto dos

processos emissores, envolvidos na comunicação. Esta característica define as diferentes formas de comunicação possíveis entre processos. Dependendo do grau de estruturação das primitivas de comunicação e sincronização as formas de identificação e endereçamento serão distintas.

III.1.3.1 IDENTIFICAÇÃO IMPLÍCITA

Num sistema, que utilize o esquema de identificação implícita, a associação entre processos comunicantes ocorre por ocasião da criação desses processos, e se mantém fixa durante toda a sua vida. GENTLEMAN [58] aponta que esta situação estática, embora usada em alguns sistemas com sucesso como no caso dos "pipes" e "filters" no contexto de UNIX [134], apresenta-se como muito limitada.

Uma variação dessa forma de identificação pode ser conseguida transformando-se as associações fixas, entre os processos comunicantes, em associações temporárias, válidas durante o tempo que for necessário. Depois, as associações podem ser alteradas para permitir uma nova situação de comunicação, permanecendo assim por um determinado tempo, como acontece, de forma análoga, para tratar a entrada/saída nas linguagens Lisp (Mc CARTHY [110]) e BCPL (RICHARDS e WHITBY-STREVENS [133]). Menos limitante que a situação anterior, esta solução é apropriada para aplicações nas quais encontra-se um processo trocando mensagens longas com vários outros processos, ou uma sequência não interrompida de mensagens curtas, com um mesmo processo.

III.1.3.2 IDENTIFICAÇÃO EXPLÍCITA

A **identificação explícita** de processos leva em conta a existência de nomes, de endereços, e de outros atributos de processos indicando explicitamente a sua localização no sistema. Essa identificação pode estar organizada de duas maneiras: direta e indireta (ANDREWS e SCHNEIDER [5]).

No caso de **identificação direta**, os processos indicados pelas primitivas são identificados pelos seus nomes ou endereços, que aparecem como entidades globais do sistema. O nome

corresponde ao identificador lógico e o endereço ao identificador físico, de forma que, para serem usados pelo sistema de comunicação, os nomes têm que passar por uma tradução, gerando os endereços.

A linguagem CSP (HOARE [68]) é um exemplo típico de utilização de identificação direta onde tanto o comando de recepção quanto o de emissão possuem indicação recíproca dos processos envolvidos.

No caso de **identificação indireta**, os processos apresentam atributos a eles associados, consistindo de portas, canais, elos ("links"), "pipes", ou nomes de procedimentos, que constituem os elementos de comunicação entre processos.

Outro exemplo é o uso do "rendezvous" de ADA (ICHBIAH et alii [73]) no qual o emissor identifica explicitamente o receptor através do ponto de entrada no comando "accept" e da tarefa à qual ele pertence. Este é o caso de ter-se um endereçamento unilateral onde o emissor indica o receptor, mas o receptor está disponível para comunicar-se com qualquer emissor.

No caso de uso de portas, que será analisado com mais detalhe numa outra seção deste capítulo, as mensagens são enviadas para portas e recebidas através das portas de diferentes maneiras. Mas a característica comum é a associação que precisa ser feita entre as portas e os processos ou módulos, e as ligações entre portas. Estas ligações podem ser feitas de forma estática, na inicialização do sistema, e de forma dinâmica, ao longo da vida do sistema. Isto traz a vantagem de se poder programar cada módulo de um sistema, sem a preocupação de especificar o nome do módulo receptor, satisfazendo os requisitos de modificabilidade e extensibilidade, e oferecendo a possibilidade de se introduzir novos módulos.

Tratamento parecido a este precisa ser implementado no caso de uso de chamada remota de procedimento para linguagens distribuídas, baseadas em módulos com interfaces explícitas de importação e exportação, e permitindo a existência de várias instâncias de cada módulo. Nestes casos precisa-se fazer uma associação entre as chamadas de procedimentos e as instâncias dos módulos aos quais estes procedimentos pertencem. Como exemplo podemos citar o caso descrito nesta tese baseado na linguagem Modula-2 com as extensões propostas.

Como exemplo de uso de identificação indireta, pode-se citar o Sistema Operacional Thoth (CHERITON et alii [37]), que usa elos ou canais entre processos, gerados e associados ao processo na sua criação e referenciados pelas primitivas de emissão e recepção. Entre outros exemplos existentes podemos citar a linguagem ARGUS (LISKOV [94], WILLIAMSON e HOROWITZ [174]) que define guardiães, contendo objetos e processos, com a restrição de que a comunicação entre processos de guardiães diferentes se dê através de portas, que são as únicas entidades globais do sistema.

A identificação direta, por sua simplicidade, torna-se adequada em configurações estáticas, enquanto a identificação indireta, por sua flexibilidade, é mais utilizada em configurações dinâmicas.

III.1.3.3 PRIMITIVAS SEM IDENTIFICAÇÃO

As primitivas de emissão e recepção sem identificação têm utilização garantida em determinados tipos de comunicação, sobretudo em sistemas de controle e sistemas de biblioteca.

A primitiva **envia-sem-identificação** ("send-any") serve para a emissão de mensagens para qualquer processo que esteja pronto e queira recebê-la ou para um conjunto de processos. Este tipo de comunicação é bastante usado em sistemas de controle [85], com o objetivo de transmitir tanto mensagens, informando o estado do sistema, quanto alarmes, indicando alguma falha, ou estado de emergência.

A primitiva **recebe-sem-identificação** ("receive-any") discutida por GENTLEMAN [58] e CUNHA et alii [44], permite ao processo que a executar, aceitar a primeira mensagem que aparecer, sem importar-se com a identidade do emissor. Tal característica é bastante útil na implementação de comunicação assimétrica, a ser analisada a seguir, facilitando a elaboração de processos gerentes de recursos e de processos de biblioteca. Cabe citar que o não determinismo na recepção de mensagens, existente nesta primitiva, torna-se responsável pela elevação do nível de paralelismo do sistema.

III.1.3.4 IDENTIFICAÇÃO SIMÉTRICA E ASSIMÉTRICA

A **identificação simétrica**, conforme MARTELLI e TARINI [109]

e AHAMAD e BERNSTEIN [2] é caracterizada pela necessidade de especificação mútua dos nomes, portas, ou procedimentos dos processos envolvidos numa comunicação, enquanto que a **identificação assimétrica** exige, somente, que o processo emissor identifique o processo receptor. O atendimento de requisições de operações, por parte do processo receptor, no caso assimétrico, é realizado sem o conhecimento dos emissores.

A linguagem CSP (HOARE [68]) constitui-se num exemplo típico do caso simétrico, e as linguagens ADA (ICHBIAH et alii [73]) e DP (BRINCH HANSEN [29]) situam-se no caso assimétrico, assim como a linguagem Menyma (KOCK e MAIBAUM [84]), na qual foi incorporada em particular a primitiva **recebe-sem-identificação** ("receive-any"), já analisada anteriormente.

Mecanismos de comunicação com identificação simétrica introduzem algumas limitações fortes na comunicação. O fato do processo receptor precisar identificar o processo emissor da mensagem que ele está esperando, não permite implementar não determinismo na ordem de recepção das mensagens a serem recebidas, a menos que sejam usados comandos guardados, como por exemplo em CSP. Isto reduz o paralelismo do sistema, uma vez que a especificação da ordem em que as mensagens devem ser recebidas pode retardar processos que estiverem prontos para se comunicar.

Outra limitação imposta por este comportamento do processo receptor é que o mesmo não pode ser usado como uma subrotina de biblioteca, ou como um servidor, por não haver possibilidade de se conhecer o nome de todos os processos que poderiam chamá-lo, exceto no caso de estes serem poucos e conhecidos.

Portanto, este tipo de endereçamento é contra-indicado em aplicações do tipo cliente/servidor, principalmente em situações onde os clientes são variáveis.

A comunicação com identificação assimétrica resolve a maioria dos problemas anteriormente levantados e, embora não satisfaça todas as aplicações possíveis, ela é adequada para aplicações do tipo cliente/servidor, pois permite implementar não determinismo na ordem de recepção das mensagens e permite o uso de rotinas de bibliotecas associadas.

Precisamos salientar que nesta forma de identificação, nos casos de comunicação bidirecional, ou no caso de precisar enviar uma resposta para outro processo, é necessário salvar o

remetente. Como exemplo, podemos citar a implementação de BRINCH HANSEN para a linguagem Pascal Concorrente [27], na qual é assegurada a área de memória na qual é recebida a chamada ao procedimento do monitor para enviar os resultados de volta.

III.1.3.5 FORMAS DE COMUNICAÇÃO ENTRE PROCESSOS

Dependendo do tipo de identificação, utilizado nas primitivas de comunicação e sincronização, é possível obter-se diferentes formas de comunicação entre os processos. Como foi citado por KIRNER em [81], é de fundamental importância, especialmente em sistemas distribuídos, analisar as formas em que os processos interagem entre si. Esses processos costumam ser utilizados para: obtenção de paralelismo; aumento de disponibilidade de dados; redução de tempo de resposta; compartilhamento de recursos e aumento de confiabilidade. Segundo BACON [12] as diferentes formas de comunicação entre processos são as seguintes:

- comunicação um para um;
- comunicação vários para um;
- comunicação um para vários;
- comunicação vários para vários.

Na **comunicação um para um**, ou de origem e destino único, só um processo emissor e um processo receptor se comunicam. As primitivas mais adequadas para este caso são as que implementam comunicação simétrica da forma como ocorre em CSP [68]. Quando forem utilizadas primitivas com identificação indireta, implementadas, por exemplo, através de portas de entrada e saída como em [102], a comunicação será obtida ligando-se uma porta de saída a uma porta de entrada. Esta forma de comunicação é particularmente utilizada em sistemas estruturados da maneira "pipe-line" ou produtor/consumidor, na qual a saída de um processo é entrada para outro processo envolvido na comunicação.

Na **comunicação vários para um**, ou de origem múltipla, vários processos de origem podem se comunicar com um processo destino. Como exemplo, tem-se o processo servidor ou gerente, ou o

processo de biblioteca descrito anteriormente, que se comunica com vários outros processos clientes. Para este caso particular, as primitivas devem permitir comunicação assimétrica, pois o processo receptor não conhece a identificação dos processos que lhe poderão enviar mensagens. Se for utilizado o endereçamento explícito indireto, o servidor poderá ser substituido sem que os clientes percebam. Se os processos emissores precisarem receber uma mensagem de resposta, como parte da execução da primitiva, esta deverá ser do tipo bloqueado. As linguagens ADA (ICHBIAH et alii [73]) e DP (BRINCH HANSEN [29]), as primitivas sugeridas por GENTLEMAN [58] e o projeto CONIC (KRAMER et alii [85]) são exemplos desse tipo de comunicação. Em outras situações, onde todos os processos que se comunicam sejam conhecidos, pode-se utilizar comunicação simétrica e primitivas do tipo bloqueado. No caso de serem utilizadas portas de entrada e saída, este tipo de comunicação será obtido pela ligação de várias portas de saída a uma mesma porta de entrada.

Na **comunicação um para vários**, ou de destino múltiplo, um processo emissor pode comunicar-se com vários processos destino. Uma forma de implementar esta comunicação, através de uma única emissão, é usar a primitiva envia sem identificação, já descrita anteriormente. Se as primitivas disponíveis usarem identificação direta, esta comunicação deverá ser baseada em várias emissões. No caso de utilização de portas de entrada e saída, este tipo de comunicação será obtido pela ligação de uma porta de saída a várias portas de entrada. Este tipo de comunicação pode ser usado para a notificação de alarmes e de condições excepcionais para várias partes do sistema. A comunicação um para todos, conhecida por difusão ("broadcasting") já mencionada anteriormente, é um caso particular deste tipo.

Na **comunicação vários para vários**, ou de origem e destino múltiplos, vários processos emissores podem comunicar-se com vários processos receptores simultaneamente. Este tipo de comunicação pode ser utilizado no caso de ter vários servidores análogos ou iguais atendendo vários clientes. Uma forma possível de implementação consiste em usar portas compartilhadas nos processos servidores, e comunicação vários para um.

III.1.4 PORTAS

O conceito de porta está sendo cada vez mais utilizado para intercomunicação entre processos, e depois de ter feito uma análise exaustiva da literatura, pudemos comprovar que ele aparece com sintaxes e semânticas muito variadas e com diferentes graus de estruturação associados.

O conceito de porta como mecanismo de comunicação foi introduzido por BALZER [13] e WALDEN [171] como forma de nomeação indireta na troca de mensagens entre processos: as primitivas de comunicação nomeiam portas que estão ligadas de alguma forma aos processos. As portas representam canais de comunicação entre os processos e mascaram a identidade dos processos envolvidos na comunicação. Este modelo, chamado endereçamento funcional, distingue a comunicação baseada em portas, de mecanismos de comunicação que usam outras formas de endereçamento.

O conceito de porta evoluiu e foi-lhe associado um tipo de dados que identifica as mensagens que podem ser transmitidas através dela, fornecendo proteção em relação aos objetos comunicados. As portas podem ser diferenciadas pelas funções que exercem como, por exemplo, portas de entrada ou de saída. Estas portas precisam ser ligadas para estabelecer a comunicação, implícita ou explicitamente através de primitivas, que permitam implementar diferentes tipos de comunicação, e portanto diversas topologias. Estas ligações podem ser de dois tipos: estáticas, ou seja, estabelecidas na inicialização do programa, ou dinâmicas, realizadas durante a execução do programa.

Dependendo se a comunicação entre processos é síncrona ou assíncrona, serão necessários diferentes tratamentos de armazenamento de mensagens asociados às portas.

Uma porta é reconhecida por um sistema como uma entidade que é independente do processo que a usa. Portas podem então ser consideradas como tipos abstratos de dados que são manipulados por conjuntos predefinidos de operações.

O conceito de porta, usado desta forma, permite introduzir modularidade. Um sistema pode ser programado como um conjunto de módulos, cada um com uma interface explícita visível, que está

definida pelas portas do módulo. A configuração do sistema é realizada através da ligação entre as portas das interfaces dos diferentes módulos.

Esta forma de explicitar e encapsular os mecanismos de comunicação entre módulos, através da definição da interface usando portas, permite abstrair a comunicação da forma de implementação. Neste caso estariam associadas às portas, através de operações, os diferentes mecanismos de comunicação com seus respectivos protocolos.

No contexto de sistemas distribuídos aparecem alguns problemas a serem resolvidos, como os de nomeação, isto é, o escopo dos nomes das portas e das ligações. O uso do conceito de porta aparece tanto na programação dos módulos do sistema distribuído quanto na fase de configuração do sistema. Como neste capítulo estamos concentrados na programação em pequena escala, os problemas de ligação serão tratados somente para os modelos que não distinguem claramente os dois níveis de programação.

III.1.4.1 ALGUMAS CARACTERÍSTICAS BÁSICAS DAS PORTAS

III.1.4.1.1 NOMEAÇÃO

Existem na literatura diferentes modelos para realizar nomeação de portas.

SILBERSCHATZ [149] acrescentou à linguagem CSP (HOARE [68]) o uso de portas na nomeação dos comandos de emissão e recepção originais, substituindo os nomes dos processos pelos nomes das portas. Os dois processos que se comunicam precisam nomear a mesma porta sendo que um processo é o dono da porta e o outro é seu usuário.

As portas são então objetos globais com nomes únicos e este esquema permite que a comunicação não seja mais de origem e destino único como em CSP, mas sim de origem múltiplo de forma não determinística. As portas não tem tipo nem direção associada.

Existem outros modelos, como o de LISKOV [94], nos quais os processos de guardiães diferentes se comunicam enviando mensagens explicitamente através de portas. Estes são os únicos objetos globais e um processo envia uma mensagem para a porta de outro processo pertencente a um guardião diferente.

As portas são unidirecionais e têm tipo associado: elas são descritas pelas mensagens que lhes podem ser enviadas. Cada porta está associada a um conjunto de mensagens possíveis, e para cada uma pode estar definida alguma resposta no caso de comunicação do tipo pedido/resposta.

As primitivas associadas a esta forma de porta são o envio não bloqueado, com especificação opcional de porta de resposta indicando a porta de outro guardião (não necessariamente quem enviou), e a recepção bloqueada contendo em geral uma lista de portas, com uma sequência de comandos associados a cada porta. Dependendo da mensagem recebida é executada a sequência do comando e opcionalmente é enviada uma resposta para a porta indicada. Isto é, existe uma estruturação no sentido de associar na recepção o código a ser executado para cada tipo de mensagem recebida. Esta primitiva inclui também o tratamento de falhas e de bloqueio. Note-se que as portas da lista pertencem ao processo que executa a recepção.

Na proposta de MAO e YEH [108], que é uma extensão da linguagem DP (BRINCH HANSEN [29]), foi acrescentado o conceito de porta, e ele é utilizado de forma análoga ao modelo de LISKOV. O processo que contém a porta é seu dono, e os outros processos fazem chamadas à porta precisando conhecer o nome do processo dono e o nome da porta.

Nestes dois esquemas analisados não é necessário fazer ligações de portas, já que a comunicação entre processos é estabelecida nomeando portas, que têm nomes globais ao sistema. Se por um lado esta forma é mais simples, ela limita enormemente a modularidade do sistema, já que cada módulo não pode ser programado independentemente dos outros e compromete a flexibilidade do sistema.

O esquema que está sendo mais utilizado nos últimos anos, e particularmente para sistemas distribuídos, é o de portas locais. Cada unidade modular, chamada processo (REID [131]), módulo (JOSEPH [77], KRAMER et alii [86]), nó (FANTECHI et alii [54], RUGGIERO e BRESSAN [137]), dependendo do modelo, define as portas locais. Isto permite que cada unidade modular possa ser programada sem precisar conhecer as outras, e na etapa de configuração serão ligadas as portas das diferentes unidades modulares para estabelecer a comunicação. Os problemas de

ligações e de visibilidade dos nomes serão tratados mais adiante.

Neste esquema cada unidade modular contém as suas próprias portas e os processos enviam e recebem mensagens através delas. No uso de portas locais existem também diferentes esquemas que vale a pena destacar.

No modelo do projeto CONIC (MAGEE et alii [106]) as portas são declaradas no nível do módulo de tarefa ("task module"), e elas representam a interface do módulo, isto é, a parte visível, para poder se comunicar com outras tarefas. Existe outro nível de estruturação, que é denominado de módulo de grupo ("group module"), composto de vários módulos tarefas ou vários módulos de grupo. Neste nível também são declaradas portas locais ao módulo de forma análoga à descrita anteriormente, e várias tarefas podem enviar ou receber mensagens da mesma porta. Estas portas são utilizadas para a comunicação entre diferentes módulos pertencentes a níveis de hierarquia maior.

No caso do modelo MCD (RUGGIERO e BRESSAN [137]) as portas são declaradas no nível do nó, mas na hora de criar os processos, elas são-lhes associadas de maneira que cada porta só pode pertencer a um processo.

No projeto Cnet (FANTECHI et alii [54]) baseado na linguagem ADA, foi acrescentado, além do conceito de nó, o de porta para a comunicação entre tarefas de diferentes nós. O nó é constituido por um conjunto de tarefas que se comunicam internamente através do "rendezvous" de ADA. Dentro de cada nó são criadas portas que serão utilizadas pelas tarefas. A visibilidade interna das portas é dada pelas regras de escopo de ADA, e elas serão utilizadas para se comunicar com tarefas de outros nós.

O método Mascot, cuja última versão Mascot3 é apresentada por BATE em [16] e por SIMPSON em [151], é utilizado para projetar e implementar sistemas. Seus componentes básicos são atividades (componentes ativos) que se comunicam através de áreas de dados de interconexão (IDA) (componentes passivos). As atividades definem portas locais que são ligadas às janelas, definidas nas IDA também localmente, através de declarações de interfaces associadas a procedimentos. As ligações entre portas e janelas são feitas através da inicialização de instâncias de atividades com os respectivos parâmetros associados às suas portas, ou seja, de forma implícita na etapa de configuração.

Portanto, dentro das atividades as portas são referenciadas através de parâmetros locais.

Existem outros modelos mais estruturados (JOSEPH [77], REID [131]), que serão tratados mais adiante, baseados na mesma filosofia de localidade.

A característica principal deste esquema é a grande modularidade que oferece, e o fato de mostrar uma interface visível explícita da unidade modular, através da definição das portas a ela associadas. As tarefas dentro do módulo, cujas funções são desconhecidas e irrelevantes ao mundo externo, são escondidas detrás da interface. Para o programador as portas são nomes locais para fontes e destinos arbitrários de mensagens. O adiamento na associação entre portas de diferentes módulos pode ser vista como uma ligação deferida, por exemplo entre uma mensagem e seu destino, do tempo de compilação para a etapa de configuração. Isto facilita modificar e estender um sistema através de reconfiguração e permite incluir módulos de biblioteca.

III.1.4.1.2 TIPOS E FUNÇÕES DAS PORTAS

Nos modelos de portas usadas para simples transmissão de mensagens, existem alguns, como por exemplo o de SILBERSCHATZ [149], nos quais a porta não está associada a nenhum tipo de mensagem nem a nenhuma função específica.

No entanto, a maioria dos modelos associa às portas os tipos das mensagens que por elas são transmitidas. Isto permite associar áreas definidas às portas, para armazenar as mensagens, e fazer testes para que a mensagem esperada coincida com a mensagem enviada, isto é, são implementados mecanismos de verificação na transmissão. Já vimos, por exemplo, que no modelo de LISKOV as portas estavam associadas a um conjunto de mensagens.

Além de pelo tipo, as portas se caracterizam pelas funções que exercem. Podemos achar na literatura portas de entrada e portas de saída unidirecionais, portas de entrada e saída e portas de difusão ("broadcasting") bidirecionais.

A maioria dos modelos (JOSEPH [77], FANTECHI et alii [54],

RUGGIERO e BRESSAN [137], SILBERSCHATZ [149], REID [131], KRAMER [86]) definem portas locais de entrada e de saída separadas, e a comunicação entre módulos é estabelecida ligando portas de entrada de um módulo com portas de saída de um outro, implementando uma abstração de canal de comunicação unidirecional. Dentro do módulo os processos ou tarefas recebem mensagens de suas próprias portas de entrada e enviam mensagens para as suas próprias portas de saída.

No modelo MCD (RUGGIERO e BRESSAN [137]) são declaradas portas de entrada e de saída unidirecionais no nível do nó, e estas são depois associadas aos processos na etapa da sua criação. Cada porta de saída precisa ser ligada a uma porta de entrada de um outro processo do mesmo nó ou de nós diferentes. A única forma de comunicação entre processos é realizada através das portas, de forma análoga, dentro do mesmo nó ou entre nós diferentes.

O projeto Cnet (FANTECHI et alii [54]) define portas bidirecionais chamadas de difusão, uma associada a cada nó, que permitem dar a visibilidade mínima da rede para cada nó no nível da linguagem. Todas as portas de difusão estão ligadas entre si e são instanciadas em cada nó na etapa de configuração. Elas são utilizadas para enviar mensagens para todos os nós da rede, tais como mensagens de controle, inicialização de canais, mensagens de erros da rede, correio, etc.

Na maioria dos casos, são combinados o conceito de tipo e de função para cada porta, de maneira que na declaração são definidas as duas propriedades.

No projeto Cnet foram acrescentadas portas locais de entrada e de saída unidirecionais, associadas aos tipos de mensagens transmitidas, para poder introduzir comunicação indireta entre nós,e fornecer teste de tipo de mensagens. Para isto impõe-se a restrição, que uma porta de saída de um nó só pode ser ligada com uma porta de entrada de um outro nó, correspondente ao mesmo tipo de mensagens.

Analogamente no caso do projeto CONIC (KRAMER et alii [86]), as portas são declaradas ou de entrada ou de saída, e associadas a um tipo de mensagem. Neste modelo é associada também à porta a característica do tipo de comunicação, isto é, se a comunicação através da porta é síncrona ou assíncrona. No caso da porta ser

síncrona, seja de entrada ou de saída, ela é considerada bidirecional, já que vai ter uma resposta associada.

Em Mascot3 as portas e janelas são associadas a tipos de interfaces, contendo a declaração de um ou mais procedimentos, e na hora de serem ligadas, seus tipos devem coincidir.

Vemos que podem ser incorporadas várias características às portas e analisaremos mais adiante as mais estruturadas que já foram mencionadas e que correspondem a níveis de abstração de dados diferentes.

III.1.4.1.3 LIGAÇÕES

Como já foi mencionado, quando a comunicação entre processos é realizada através do uso de portas, é necessário que estas portas sejam ligadas para poder estabelecer os canais de comunicação. Trateremos aqui das ligações explícitas feitas entre portas do mesmo ou de diferentes módulos, isto é, para os módulos que tratam as portas localmente, já que no caso das portas serem globais, não existem ligações explícitas.

Estas ligações representam a fase de configuração do sistema a partir de seus componentes individuais e poderão ser feitas em diferentes momentos, seja na inicialização do sistema, seja na etapa de compilação de forma estática ou seja na etapa de execução de forma dinâmica.

Serão analisados somente aqueles casos nos quais os dois níveis de programação em pequena escala e em larga escala não estão nitidamente separados.

No caso do modelo MCD (RUGGIERO e BRESSAN [137]), a primitiva de enlace liga uma porta de saída a uma porta de entrada e pode ser executada entre os comandos de inicialização do nó para estabelecer enlaces iniciais entre os processos do mesmo nó ou de nós diferentes. Não existe controle de tipo associado à porta, à diferença de outros modelos, e queremos destacar que as ligações são feitas entre dois únicos processos, isto é, entre duas portas. Uma porta de saída só pode ser ligada a uma porta de entrada, embora várias portas de saída possam ser ligadas à mesma porta de entrada através da criação de vários enlaces distintos.

Aqui aparece o problema do conhecimento de nomes de portas

de outros nós e existem várias formas de tratá-lo. Vários modelos, como por exemplo o MCD, tornam os nomes das portas visíveis aos outros nós através da sua exportação e importação. Estes modelos diferem entre si da seguinte forma: para alguns é necessário que os módulos exportem as suas portas, de forma que as tornam visíveis aos demais módulos, e não é necessária importação explícita; outros exigem que haja exportação e importação explícita, e que as ligações só possam ser feitas quando haja declarações correspondentes. Modelos que implementam maior proteção ainda, exigem que, na exportação, seja indicado para quem se exporta, e, na importação, de quem se importa, isto é, que se associem os módulos às portas exportadas e importadas.

Vemos que os vários modelos oferecem diferentes graus de proteção limitando os direitos de acesso às portas de maneiras distintas.

No projeto Cnet, as portas de entrada e saída são criadas localmente nos nós, e somente para as portas de saída são criados nomes externos, que serão gerados por um servidor de nomes e utilizados na ligação através da primitiva BIND. O nome da porta de entrada será visível fora do nó através do nome externo, e os outros nós, que desejem se comunicar com o nó dono daquela porta, ligarão as suas portas de saída através da primitiva CONNECT com essa porta de entrada. Neste modelo é possível fazer ligações de um processo se comunicando com um ou vários outros processos.

Em Mascot3 podem ser ligadas várias portas a uma mesma janela, mas somente uma janela pode ser ligada a uma porta. Isto é, várias atividades podem se comunicar através da mesma IDA com outras atividades.

É importante caracterizar em cada modelo que tipo de ligações poderão ser estabelecidas, para identificar as diferentes topologias de comunicação possíveis. Dependendo dos modelos, os diferentes tipos de ligações são: comunicação entre processos do tipo ponto a ponto (um a um), tipo cliente/servidor (vários para um), tipo difusão (um para vários) e do tipo cliente/multiservidores (vários para vários).

III.1.4.1.4 PRIMITIVAS DE TRANSMISSÃO E RECEPÇÃO

Neste esquema de uso de portas para troca de mensagens, não existe muita variedade no tipo de primitivas definidas; os processos enviam mensagens para as portas de saída e recebem mensagens das portas de entrada.

No caso mais simples visto anteriormente (SILBERSCHATZ [149]) só existem comandos de entrada e de saída.

No caso de LISKOV, também, já foram analisados o envio assíncrono para a porta do processo do outro guardião, e a recepção síncrona sobre uma lista de portas, introduzindo certo não determinismo. Nos dois casos existe a opção de indicar uma resposta para alguma porta, seja do guardião que enviou, ou de outro. A primitiva de recepção por ser bloqueada e oferecer tratamento de falhas e saída por temporização.

No modelo MCD, as primitivas são muito parecidas às definidas por LISKOV, embora apresentem algumas diferenças, descritas a seguir. As primitivas utilizadas em um processo podem se referir apenas às portas definidas no nó onde o processo foi ativado, já que as portas são locais ao nó. A primitiva de envio é síncrona, bloqueando o processo até ser verificado o sucesso da transmissão, e portanto contém um tratamento opcional de falhas; a primitiva de recepção pode ser síncrona ou assíncrona.

No projeto CONIC, existem dois tipos de primitivas, dependendo se a comunicação é síncrona ou assíncrona. Para comunicação síncrona estão definidas as primitivas "send-wait" e "receive-reply", que implementam uma transação parecida com o "rendezvous" de ADA, com a diferença que a saída por temporização da primitiva de envio é sobre o término da execução da chamada na tarefa fonte. Para a comunicação assíncrona é fornecido o conjunto "send-receive", unidirecional, potencialmente multidestino, que permite transferência de alarmes e de informação de estado.

No caso do projeto Cnet, a primitiva de envio para a porta local de saída é assíncrona, e pode ser gerada uma exceção no caso da porta não estar ligada. A primitiva de recepção depende do valor do parâmetro de tempo associado, podendo ser assíncrona, bloqueada por tempo limitado ou bloqueada eternamente. Existe a possibilidade de receber num conjunto de portas de entrada de

forma não determinística.

III.1.4.2 SISTEMAS DINÂMICOS

Uma das principais vantagens que o uso de portas oferece é a possibilidade de escrever sistemas dinâmicos, já que permite fazer ligações de forma dinâmica entre as portas em sistemas dos quais os componentes podem ser criados e destruidos durante sua execução. Como no ítem anterior sobre o tema das ligações, aqui também só trataremos dos casos nos quais os dois níveis de programação não estão nitidamente separados.

No modelo de LISKOV, o mapeamento dos nós lógicos (guardiães) nos nós físicos é responsabilidade do programador, e existe criação e destruição dinâmica de guardiães. Os guardiães são criados por um processo de um guardião do mesmo nó físico, respondendo a mensagens de pedidos, recebidos de guardiães de outros nós. Quando um guardião é criado, as suas portas são conhecidas para o processo que o criou, já que as portas têm nomes globais. Os nomes das portas podem ser enviadas nas mensagens, de forma que várias mensagens de diferentes fontes podem chegar à mesma porta.

No modelo MCD, os nós lógicos são entidades fixas, que são alocadas aos nós físicos na fase de configuração. Os processos são declarados dentro dos nós, e são criadas instâncias deles por comandos executados na inicialização do nó, ou dentro de outros processos. A cada criação de instância de processo são associadas as portas declaradas no nível do nó de forma fixa, isto é, existe um número máximo de portas, e elas vão sendo associadas aos processos, de forma que cada processo tenha portas distintas.

As ligações estáticas de portas são feitas na inicialização do nó, mas podem ser feitas ligações dinâmicas no nível dos processos, depois da criação de um processo, e quando for recebida uma porta através de uma mensagem. Este modelo não apresenta primitivas explícitas para destruir ligações nem processos, que parecem acabar somente quando recebem uma mensagem dando essa ordem.

No projeto Cnet, os nós lógicos são fixos e são alocados estaticamente aos nós físicos. Cada nó está formado por unidades de programa padrão de ADA, e portanto existem criação e

destruição dinâmicas de tarefas. Dentro de cada nó são criadas localmente portas, que são objetos padrão de ADA, e não são visíveis externamente. Para poder estabelecer comunicação entre os nós através das portas, é preciso que algumas delas sejam associadas a objetos externos, de forma que conexões indiretas possam ser estabelecidas, através destes objetos, entre nós que os compartilham. O nome externo da porta representará então a abstração da ligação física entre os nós que se comunicam.

A dinâmica da comunicação do sistema é implementada através da criação e destruição explícitas de portas, da ligação e do desligamento entre o objeto interno e o nome externo para as portas de entrada, e a conexão e desconexão entre as portas de saída locais de um nó com o nome externo da porta de entrada do outro nó.

Os exemplos analisados mostram as diferentes formas que o conceito de porta oferece para implementar reconfiguração dinâmica, requisito muito importante para sistemas distribuídos.

III.1.4.3 USO DE PORTAS EM MODELOS ESTRUTURADOS

Depois de ter analisado as características principais associadas ao conceito de portas, através do estudo de alguns exemplos de portas utilizadas para implementar troca de mensagens, apresentaremos nesta seção noções mais estruturadas, ligadas ao conceito de porta, que permitem esquematizar e modelar sistemas distribuídos, abstraindo-se dos detalhes da implementação dos diferentes tipos de comunicação.

LINGUAGEM ADA

A linguagem ADA, projetada para programar sistemas distribuídos, é às vezes considerada como uma linguagem que usa a noção de portas: no sentido que as "entry" por ela definidas e exportadas seriam análogas a portas de entrada dentro do programa sequencial de cada tarefa. Isto é, a tarefa define os pontos nos quais aceita chamada de outras tarefas, e nos quais executará comandos, ficando as duas tarefas sincronizadas, até que a tarefa chamada devolva algum resultado e libere a tarefa chamadora.

Neste esquema, a nomeação é direta. Não existe ligação

explícita, e nem proteção em relação a para quem se exporta e de quem se importa. As ligações são feitas através das regras de escopo, que tornam as "entry" visíveis, e através do fato que as tarefas podem ser passadas como parâmetros. Não fica explícito quais são as topologias implementadas, e ADA permite programar sistemas dinâmicos, mas de forma implícita. Neste caso pode ser feita nomeação indireta também, através das variáveis do tipo tarefa, cujos valores são passados nos parâmetros.

LINGUAGEM "COMMUNICATION PORT"

Outra linguagem, que usa a noção de porta de forma muito similar, é a de MAO e YEH [108] que está baseada na linguagem DP (BRINCH HANSEN [29]). MAO introduziu o conceito de porta de comunicação para acrescentar duas características a DP: a porta de comunicação especifica precisamente que processos podem se comunicar (parecido com ADA), assim como quando a comunicação entre os processos deve ser cortada através de um comando de desconexão que pode passar parâmetros; esta última característica aumenta o paralelismo dos programas concorrentes.

LINGUAGEM *MOD

A linguagem *MOD (COOK [42]), que foi projetada para programação distribuida, está estruturada em módulos; no módulo da rede estão especificadas de forma estática as ligações entre os módulos processadores, que podem pertencer a diferentes processadores físicos, e que por sua vez podem conter outros módulos, como, por exemplo, o módulo comunicador. *MOD está composta de processos, portas e procedimentos, que são referenciados com a mesma sintaxe; ela só fornece portas e processos para a comunicação entre processadores.

As portas estão associadas a operações a serem executadas, que podem ou não devolver resposta, e estas portas são especificadas no começo dos módulos, e são exportadas para os outros módulos, através de comandos explícitos.

Os corpos das portas são declarados dentro do processo comunicador correspondente a um determinado módulo, e cada um é definido por uma região a ser executada. Estas portas serão

chamadas, da mesma forma que chamadas de procedimentos, por procedimentos pertencentes a processos do mesmo ou de outros módulos, no caso de terem sido exportadas. Poderão se formar filas de chamadas associadas a cada porta, e o processo comunicador as atenderá de forma não determinística.

Os processos se comunicam através das chamadas das portas pertencentes ao processo comunicador, que fica permanentemente ativo. As portas não estão incluidas dentro dos outros processos, como no caso de ADA, onde as "entry" poderiam ser consideradas análogas às portas de *MOD, no sentido de ser pontos de entrada: nesta linguagem o processo comunicador é o gerente das portas.

O conceito de porta foi incluido para permitir maior flexibilidade nos tipos de comunicação a serem implementados. Além da forma chamada/resposta, era desejável incluir co-rotina, "rendezvous" e gerenciadores de diversos protocolos. A porta é vista como uma fila de mensagens independente do gerenciador, que é quem indica a execução da operação associada a ela.

A linguagem *MOD não permite projetar sistemas dinâmicos, já que não fornece primitivas de criação e destruição de componentes, nem possibilidade de fazer ligações dinâmicas. As ligações entre módulos são feitas no início na etapa de configuração, e as portas são visíveis externamente, através das regras de escopo internas dos módulos e das primitivas explícitas de exportação e importação. As portas representam a interface de comunicação entre os processos dos diferentes módulos.

ESPECIFICAÇÃO ESTRUTURADA DE SISTEMAS DE COMUNICAÇÃO

Na proposta de BOCHMANN e RAYNAL [25] o ponto central foi propor uma ferramenta de projeto para poder especificar sistemas distribuidos através de **refinamento por passos**. O método está baseado nos conceitos de processos e portas. O processo é a entidade que realiza algum processamento de dados, e é considerado a unidade de especificação. A porta é uma parte do processo, e serve para a comunicação entre ele e os outros processos do sistema. Um processo pode possuir várias portas, para se comunicar com diferentes partes do seu ambiente. A especificação das propriedades de um processo, ou de uma porta, é dada num nível abstrato, no sentido que somente o comportamento

visível externamente é descrito, mas não a forma na qual esse comportamento é realizado pela estrutura interna do processo ou da porta. As implementações dos processos e das portas são especificadas separadamente como elementos para um passo no sistema de descrição por passos.

A especificação de uma porta consiste de duas partes:

a) uma enumeração dos tipos de interações que podem ser realizadas na porta, isto é, o conjunto de operações a ela associado;

b) propriedades adicionais da porta, que podem em particular restringir a ordem de execução das operações enumeradas, ou restrições sobre os valores dos parâmetros das operações a serem executadas.

Uma porta tem um tipo associado e só pode ser ligada a portas do mesmo tipo, e é na especificação do processo onde são definidas as formas de execução das operações.

A característica importante desta proposta é seu objetivo de especificação por passos. Este refinamento pode chegar até os níveis mais baixos ligados ao circuito eletrônico.

A porta tem aqui os objetivos, de, por um lado, ser uma abstração dos pontos de entrada do sistema distribuido, e, pelo outro, permitir a especificação por passos do sistema.

MÉTODO MASCOT3

No modelo de Mascot3, as portas estão associadas aos componentes ativos, chamados atividades, e as janelas aos componentes passivos, chamados IDA. Mas os IDAs também podem possuir portas para se comunicar diretamente com outros IDAs. Estes conceitos de portas e janelas são utilizados para esquematizar a comunicação através de procedimentos entre os componentes ativos, isolando estes procedimentos em interfaces explícitas. Este esquema é utilizado tanto para comunicação entre atividades no nível mais baixo, quanto para comunicação entre subsistemas, que são compostos de atividades e IDAs, podendo estruturar o sistema em diferentes níveis hierárquicos. Mas a característica mais saliente é a separação que existe entre a

definição de uma interface e a declaração da sua implementação num componente do tipo IDA. Podem existir várias implementações para uma mesma interface, e um módulo de implementação pode implementar várias interfaces ou partes de interfaces. Este método é tratado com mais profundidade no Capítulo VII.

CONTROLE E COMUNICAÇÃO EM SISTEMAS DE PROGRAMAÇÃO

Em [131], REID apresenta um modelo para descrever mecanismos abstratos de comunicação. Este modelo de comunicação, que consiste de objetos primitivos e operações de comunicação, tem como objetivo estudar as propriedades de comunicação de sistemas. O sistema considerado é dinâmico, permitindo criação e destruição de seus elementos.

Foram focalizados dois problemas ligados à comunicação: como definir um mecanismo abstrato de comunicação, e até onde o modelo de comunicação utilizado pelo componente pode ser mudado, sem modificar o componente.

O propósito, é então, poder desenvolver uma noção de **controle abstrato de comunicação** análoga à noção de tipo abstrato de dado. Por outro lado, é importante poder adiar ao máximo a decisão de quais mecanismos deverão ser utilizados na interface entre os módulos do sistema. A flexibilidade do sistema dependerá da possibilidade de usar diferentes mecanismos de comunicação para essa interface. Desta forma um componente escrito para um contexto poderá ser usado num outro contexto. Estas questões estão relacionadas ,e precisam ser enfrentadas, para que se possa desenvolver um modelo, que seja independente de linguagens específicas, de arquiteturas e de implementações.

As restrições ,que definem um mecanismo de comunicação, expressam as propriedades essenciais que devem ser previstas por qualquer implementação. A flexibilidade é introduzida como um critério para escolher qual, entre diferentes implementações de mecanismos de comunicação, se deveria desenvolver.

O modelo proposto contém dois tipos de objetos: componentes e portas. Os componentes estão associados a programas sequenciais. O sistema programado consiste de uma coleção de componentes, que se comunicam entre si através de portas. Os objetos têm nomes únicos, e são conhecidos pelo sistema. Qualquer

componente pode operar sobre qualquer objeto do qual possui o identificador, ou por tê-lo criado, ou porque foi-lhe comunicado por outro componente. Este esquema de nomeação é escolhido para poder ser utilizado por qualquer tipo de linguagem. Cada porta tem um dono, que é o componente que a criou. As portas se caracterizam pelo subtipo: síncronas ou assíncronas, e pela direção. As portas síncronas podem ser de entrada ou saída. As portas assíncronas podem ser de entrada, de saída ou de entrada/saída. Elas não têm tipo associado às mensagens que elas transmitem.

Existem duas operações para criar e destruir componentes, e quatro operações para criar e destruir portas e ligar e desligar portas de uma em uma. O modelo fornece então criação e destruição dinâmica de objetos, e ligações de forma explícita. As portas, locais aos componentes, são diferenciadas pelas suas funções de entrada e saída, e as operações síncronas são unidirecionais. É mostrado como, com estas definições básicas, pode ser implementada uma variedade de formas de comunicação, oferecidas pelas diferentes linguagens de programação de alto nível. Para isto são introduzidas também formas de restringir a ordem das operações a serem executadas na comunicação, para poder facilitar a estruturação das construções abstratas a partir das primitivas anteriores. Qualquer conjunto de componentes, cujo comportamento satisfaça as restrições que definem uma construção abstrata de comunicação, é uma implementação aceitável. O critério de seleção de uma determinada implementação para uma forma de comunicação, neste modelo, é o de flexibilidade, e não de facilidade ou eficiência, que são dependentes da arquitetura.

A implementação será distribuida entre os vários componentes, e a sua flexibilidade dependerá da distribuição do conhecimento e das operações entre os componentes, do sistema e do usuário. O conjunto das operações realizadas pelo componente de usuário, que fazem parte da implementação de uma construção de comunicação, é chamado de interface do usuário para essa construção. Se a interface de usuário para diferentes implementações é idêntica, essas implementações são completamente compatíveis.

Este modelo permite concentrar-se nas propriedades de uma simples forma de comunicação, ignorando outros aspectos do

programa, e facilita a comparação de diferentes formas de comunicação como foi mostrado na análise do trabalho sobre dualidade (LAUER e NEEDHAM [90]), que será discutido na próxima seção .

ESQUEMAS DE COMUNICAÇÃO

Outro modelo, ainda para analisar esquemas de comunicação diferentes usando o conceito de porta, é o de JOSEPH [77]. O autor descreve uma linguagem para programação paralela e distribuida, que provê flexibilidade na transferência de informação e controle entre os componentes individuais de um programa. Os programas estão formados por módulos e o comportamento de cada módulo está determinado unicamente pelos tipos de comunicação nos quais ele participa. Dois módulos podem se comunicar de diferentes maneiras, dependendo das necessidades da informação que eles trocam entre si. Isto é realizado definindo portas ligadas de forma explícita, e associando a elas protocolos de controle.

O autor chama cada implementação de porta, **esquema de comunicação**, e considera a declaração de porta como uma instanciação do tipo de porta. A eleição do tipo de implementação da porta não afeta as instruções do módulo, e a escolha do esquema de comunicação será feita nas especificações do módulo.

Uma das vantagens desta linguagem é sua implementação eficiente numa variedade de arquiteturas, tanto para aquelas com memória compartilhada, quanto para sistemas distribuidos que só compartilham canais de comunicação. As únicas características da linguagem, que são particularmente dependentes da arquitetura, são os mecanismos de comunicação. As portas não são essencialmente diferentes dos outros tipos de dados programados na linguagem, tornando possível para o usuário definir novos esquemas de comunicação, distintos daqueles sugeridos pela implementação da linguagem. É então possível que diferentes implementações estejam presentes ao mesmo tempo, e que a especificação do módulo inclua a escolha do esquema de comunicação a ser utilizado para sua implementação. A separação entre o projeto do esquema de comunicação e o uso do mecanismo de comunicação têm o efeito colateral conveniente de que problemas,

como o de gerenciamento de áreas de armazenamento associadas, ficam ligadas ao método de implementação do esquema de comunicação, e não aos módulos que usam este esquema.

Os programas estão formados por módulos; cada um está composto de uma parte de especificação, que define as formas nas quais o módulo se comunica com os outros módulos, e uma parte de implementação, que contém as instruções das operações executadas pelo módulo. O bloco principal do módulo consiste de um conjunto de comandos que são de dois tipos, os comandos básicos e os de comunicação.

A característica mais importante que esta linguagem oferece é poder associar diferentes implementações de portas nos módulos, dependendo dos requerimentos particulares. Isto permite satisfazer diferentes necessidades de comunicação, sem acrescentar à linguagem novas características e construções. A maioria das linguagens propostas para sistemas distribuidos, ou para processamento paralelo, parece ser projetada tendo em mente uma determinada arquitetura. O uso de diferentes implementações de portas faz com que a linguagem possa ser independente de uma arquitetura particular, e que ela seja capaz de ser usada eficientemente numa variedade de arquiteturas.

A vantagem deste modelo é que pode ser usado para manipular dispositivos, para sistemas com diferentes processadores e memórias, e para sistemas com configuração hierárquica, onde podem coexistir diferentes tipos de comunicação.

O objetivo de usar os esquemas de comunicação é então separar a implementação da lógica do módulo, da escolha do esquema de comunicação a ser utilizado.

III.1.4.4 CONSIDERAÇÕES FINAIS

Foram mostradas as vantagens da nomeação indireta e funcional que o uso das portas oferece, assim como a flexibilidade para estabelecer diferentes tipos de conexões adequadas a topologias variadas. O uso de portas locais aos módulos ou processos permite aumentar a modularidade dos programas a serem projetados, e facilita a sua escrita, manutenção e modificabilidade.

Os direitos de acesso a operações de módulos podem ser

controlados, através da visibilidade das portas para os outros módulos, seja através de primitivas de exportação e importação, ou seja, através de nomes globais, que são fornecidos somente através de mensagens a determinados módulos. Podem ser implementados testes de consistência em relação às mensagens associando tipos às portas e controlando as conexões entre portas do mesmo tipo.

As portas facilitam o projeto de sistemas dinâmicos, tanto em relação à possibilidade de sua criação e destruição, quanto em relação à facilidade de fazer conexões dinâmicas. Em vários modelos analisados, mesmo com diferentes graus de estruturação, pudemos constatar como o conceito de porta permite estabelecer interfaces explícitas entre os módulos que compõem um determinado sistema. Nos últimos dois modelos apresentados aparece o conceito de porta como uma ferramenta muito poderosa para definir mecanismos abstratos de comunicação de processos para diferentes linguagens, arquiteturas e implementações. A porta pode ser considerada como um tipo abstrato de comunicação, que dependendo da sua definição pode corresponder a diferentes implementações, sendo associada a vários esquemas de comunicação.

O conceito de porta se apresenta como uma ferramenta tão útil, nos diferentes graus de estruturação apresentados, que é possível considerar o seu uso, sem precisar criar novas linguagens. Como exemplos, podemos citar o que está sendo feito pelo grupo do projeto Cnet para a linguagem ADA, e o que é sugerido no trabalho de KATAGUINI [78] sobre o sistema operacional UNIX, analogamente à idéia implementada por BIRRELL e NELSON [19, 20], para chamadas remotas de procedimentos.

Em particular, queremos destacar que o conceito de porta foi analisado tão extensamente, justamente por ter muitos pontos em comum com o conceito de chamada remota de procedimento, na sua forma mais estruturada, como foi mostrado nesta seção. Esta análise foi da maior utilidade para a escolha, definição e implementação das extensões propostas a Modula-2 neste trabalho.

III.2 ANÁLISE DA DUALIDADE DAS FERRAMENTAS DE COMUNICAÇÃO E SINCRONIZAÇÃO UTILIZADAS NA CONSTRUÇÃO DE PROGRAMAS CONCORRENTES

Neste capítulo foram analisados vários mecanismos de comunicação e sincronização para a construção de programas concorrentes a serem executados em diferentes arquiteturas, isto é, com um ou mais processadores e com ou sem memória compartilhada. Foram também apontadas algumas vantagens e desvantagens do uso destes mecanismos para as diferentes arquiteturas consideradas.

Nosso objetivo agora é expor e discutir as idéias de LAUER e NEEDHAM [90] sobre a dualidade do uso destes mecanismos. Os autores separam estes mecanismos em dois modelos que são usados para combinar os conceitos de paralelismo, sincronização e comunicação, que definem um estilo de programação. Um modelo está baseado em chamadas de procedimentos, que corresponde ao uso do conceito de monitor para controlar acesso a variáveis compartilhadas. No caso de sistemas distribuídos o modelo pode ser estendido com o uso de chamadas remotas de procedimentos (RPC). Uma RPC é feita ao nó físico no qual está carregado o monitor, por processos localizados em nós diferentes. O outro modelo, baseado na troca de mensagens, utiliza a transferência explícita de dados, seja entre processos de um mesmo nó, seja entre processos localizados em nós físicos diferentes.

III.2.1 DESCRIÇÃO DOS MODELOS ADOTADOS

LAUER e NEEDHAM demonstraram, que sob certas restrições, que serão apresentadas a seguir, existe uma dualidade entre os dois modelos, e que um programa realizado segundo um modelo tem seu correspondente direto segundo o outro. Esta dualidade pode ser usada como ferramenta na estruturação de programas concorrentes por possibilitar especificar o projeto de um sistema usando um dos modelos, e realizá-lo usando o modelo dual, como foi discutido por STANTON em [155].

Para caracterizar a sua classificação, os autores construiram um modelo canônico para cada categoria.

O **modelo baseado na troca de mensagens** é caracterizado por:

i) um número pequeno e relativamente estático de processos grandes;

ii) um conjunto explícito de canais de mensagens entre eles;

iii) uma quantidade relativamente limitada de dados compartilhados;

iv) uma identificação dos processos com o seu contexto de execução.

Neste modelo, os processos estão geralmente associados com os recursos do sistemaa, e os serviços ,oferecidos pelas aplicações que os usam, são solicitados através de pedidos, codificados em dados que são passados nas mensagens.

O modelo baseado em procedimentos é caracterizado por:

i) um número grande de processos pequenos;

ii) criação e destruição rápida de processos;

iii) comunicação por compartilhamento direto de dados na memória;

iv) identificação do contexto da execução com a função que está sendo executada mais do que com o processo.

Neste modelo os recursos do sistema são geralmente codificados em estruturas globais de dados, e as aplicações estão associadas a processos, cujas necessidades são expressas através de chamadas de procedimentos do sistema, que têm acesso a esses dados.

Estes dois modelos definem dois tipos de operações primitivas para gerenciar os processos, a comunicação e sincronização, e, a partir da experiência de seu uso, os autores chegaram às seguintes observações:

i) os dois modelos são duais, isto é, um programa ou subsistema, construído estritamente de acordo com as primitivas definidas por um modelo, pode ser transformado diretamente num programa ou subsistema dual que se ajusta ao outro modelo;

ii) os programas ou subsistemas duais são semanticamente idênticos;

iii) o desempenho do programa ou subsistema de um modelo, medido pelo comprimento das filas, os tempos de espera, velocidade de serviços etc., é idêntico ao do sistema dual, usando políticas de escalonamento idênticas.

A conclusão principal que pode ser tirada a partir destas observações é que a escolha do modelo a ser adotado não depende das aplicações às quais o sistema deve dar suporte. Ela depende basicamente do nível inferior sobre o qual o sistema é construido. Em outras palavras, as limitações impostas pela arquitetura e o hardware utilizados ou pela máquina virtual, numa arquitetura em vários níveis, definirão a escolha de qual será o conjunto de operações primitivas e de mecanismos mais adequados e mais fácil de serem implementados.

Vejamos agora de forma resumida as primitivas dos modelos que os autores definiram e seus equivalentes, sobre os quais basearam a conclusão de dualidade.

O **modelo baseado na troca de mensagens** é constituido por um número quase constante de processos, ligados através de canais de mensagens e diversas portas, nas quais são enfileiradas mensagens de diferentes tipos. Um canal de mensagem é uma estrutura abstrata que identifica o destino de uma mensagem, e uma porta pode ter mais de um canal ligado a ela. Os processos se comunicam e se sincronizam através de quatro operações de transmissão de mensagens que serão executadas pelos processos emissores e receptores.

O processo emissor envia uma mensagem para o processo receptor, que a enfileira na porta correspondente ao tipo da mensagem, ligada ao canal de mensagem indicado como parâmetro, através da seguinte operação:

SendMessage[messageChannel,messageBody]**returns**[messageId]

O processo emissor pode ficar bloqueado esperando por uma resposta para uma mensagem específica enviada pela primitiva anterior, executando a seguinte operação:

AwaitReply[messageId]**returns**[messageBody]

O processo receptor, que é o dono das portas, pode ficar bloqueado esperando receber uma mensagem através de qualquer elemento de um subconjunto de portas, especificado na operação de recepção seguinte:

WaitforMessage[setofmessagePort]**returns**[messageBody,messageId, messagePort]

O processo receptor envia uma resposta ao processo emissor do qual recebeu a última mensagem, e que está bloqueado esperando por ela, através da operação:

SendReply[messageId,messageBody]

Podemos notar que neste modelo uma mensagem é respondida antes da recepção da próxima mensagem. A relação entre o emissor e o receptor é assimétrica, já que o receptor desconhece a identidade do emissor. Devemos notar ainda que quando o emissor se bloqueia esperando a resposta, este bloqueio ocorre na fila de mensagens da porta do receptor e portanto antes da recepção da mensagem.

Os processos precisam ser declarados para depois serem criadas suas instâncias através do comando **CreateProcess**, que faz também as ligações dos canais de mensagens referenciados às portas de mensagens dos processos já existentes.

O **modelo baseado em chamada de procedimentos** está caracterizado pelos mecanismos de proteção e endereçamento sobre os procedimentos, e pelas operações eficientes de chamadas de procedimentos que levam um processo rapidamente de um contexto para outro. Este modelo já foi analisado anteriormente na apresentação dos mecanismos de comunicação e sincronização para construção de programas paralelos.

O modelo canônico contém primitivas para chamadas síncronas e assíncronas de procedimentos. As chamadas síncronas são análogas às chamadas de procedimentos nas linguagens Algol (NAUR [116]), Pascal (WIRTH [175]) ou MESA (MITCHELL et alii [113]), Pascal Concorrente (BRINCH HANSEN [28]) e Modula (WIRTH [176]).

O mecanismo de chamada assíncrona de procedimento é representado pelos comandos **FORK** e **JOIN**, definidos da seguinte maneira:

processId<--FORK procedureName[parameterList]

Este comando inicia a execução do procedimento como um processo novo com seus próprios parâmetros. O procedimento opera no contexto da sua declaração como se ele tivesse sido chamado de forma síncrona, mas o processo tem a sua própria pilha de chamada e seu próprio estado. O processo que faz a chamada, continua a sua execução, em paralelo com o outro, a partir do comando seguinte ao **FORK**, e o identificador do processo retornado pelo **FORK** é utilizado no comando seguinte:

[resultList] <-- **JOIN** processId

Este comando causa a sincronização do processo que o executa com a terminação do processo nomeado pelo identificador de processo. Os resultados deste processo são passados ao processo que o chamou, da mesma maneira que aconteceria com uma chamada de procedimento convencional. O processo criado é destruido, e a execução continua no processo que executou o **JOIN** a partir do comando seguinte a este.

Este modelo inclui também os conceitos de módulo e monitor, já vistos anteriormente, e as primitivas **NEW** e **START** para criar e inicializar novas instâncias de módulos e monitores. São utilizadas, dentro dos monitores, variáveis de condição, para implementar sincronização e exclusão mútua da execução dos procedimentos do monitor. Neste modelo, uma variável de condição tem uma fila de processos a ela associada e duas operações definidas sobre ela:

WAIT conditionVariable

Esta operação suspende o processo que a executou, o coloca na fila associada à variável de condição e libera a exclusão mútua do monitor.

SIGNAL conditionVariable

Esta operação ativa um processo que foi suspenso pela operação anterior na variável de condição, para continuar a sua execução a partir do comando seguinte, assim que ele puder obter exclusão mútua do monitor.

Quando um sistema é construido baseado neste modelo, o único

meio de interação entre seus componentes é procedimental. Os processos mudam de um contexto para outro além das fronteiras dos módulos por meio dos mecanismos de chamadas de procedimentos, e o uso de chamadas assíncronas estimula a atividade concorrente.

III.2.2 DUALIDADE DOS MODELOS

A partir dos dois modelos canônicos, os autores derivaram a seguinte correspondência entre os mecanismos básicos de cada estilo ilustrada na Figura (III.1).

Usando esta correspondência, os programas escritos baseando-se estritamente no conjunto de primitivas de um modelo podem ser transformados através de seus correspondentes no outro modelo, obtendo programas totalmente análogos, sem afetar nem a lógica nem o desempenho do programa. Os autores enfatizam que as suas afirmações só valem quando é respeitado estritamente o estilo por eles definido para cada modelo.

Sistema orientado em mensagens	<--->	Sistema orientado em procedimentos
Processos, **CreateProcess**	<--->	Monitores, **NEW/START**
canais de mensagens	<--->	identificadores de procedimentos externos
portas de mensagens	<--->	**ENTRY** identificador (procedimentos vistos fora do monitor)
SendMessage;AwaitReply (imediato)	<--->	chamada de procedimento
SendMessage;...AwaitReply (com intervalo)	<--->	**FORK;...JOIN**
SendReply	<--->	**RETURN** (de procedimento)
laço principal do gerenciador de recurso, comando **WaitForMessage**, comando **case**	<--->	"lock" do monitor, atributo **ENTRY**
ramos do comando **case**	<--->	**ENTRY** declarações de procedimentos
espera seletiva de mensagens	<--->	variáveis de condição, **WAIT,SIGNAL**

FIGURA III.1 - CORRESPONDÊNCIA DERIVADA DO PRINCÍPIO DE DUALIDADE

Podemos notar, por exemplo, que no modelo baseado na troca de mensagens, o comando **SendMessage** seguido do comando **AwaitReply** executado um tempo depois, equivale no outro modelo à criação de um processo que executará em paralelo através dos comandos **FORK** e

JOIN, este último destruindo o processo recém criado. É interessante destacar o uso do termo processo nos dois modelos de forma diferente, já que existe também a correspondência entre os processos num modelo e os monitores no outro. Em particular, no exemplo que eles apresentam de um gerenciador de recursos, os ramos do comando **case** no modelo orientado em mensagens correspondem às declarações de procedimentos vistos fora do monitor de forma semelhante ao **select** da linguagem ADA. O código que representa o laço ("loop") principal do processo, com seu **WaitForMessage** e o comando **case**, realiza exatamente a mesma função que a exclusão mútua no acesso ao monitor, ou seja, a serialização dos pedidos de serviços que são atendidos um de cada vez. O comando **case** seleciona o atendimento dos serviços requisitados da mesma forma que o fazem os diferentes nomes dos procedimentos do monitor.

Vemos então que no exemplo apresentado pelos autores no trabalho já citado, através do uso da correspondência, não são tocadas nem arrumadas as partes importantes do programa. Os algoritmos não são modificados, as estruturas de dados não são substituidas, as estratégias de interface não são afetadas. Somente o texto que representa as interações entre os componentes do sistema de clientes é modificado, e isto reflete unicamente os detalhes sintáticos das primitivas do sistema no outro modelo. A semântica do conteudo é invariante.

Os autores colocam ainda que é possível embutir o conceito de processos e as primitivas de sincronização do modelo orientado em mensagens em linguagens com forte controle de tipo, como por exemplo a linguagem MESA (MITCHELL et alii [113]). Nesse caso a dualidade dos programas seria textualmente idêntica, a menos de algumas palavras chaves da linguagem.

No final do trabalho é mencionada a controvérsia que existe sobre este tema, e os autores reconhecem que as suas conclusões não são facilmente aceitas por um conjunto razoável de pesquisadores, devido ao fato que o universo do discurso não é composto de objetos naturais, e sim de objetos feitos pelo homem. São necessárias muitas argumentações e experimentos que comprovem estas idéias, e eles mesmos finalizam o artigo dizendo que só o tempo dirá se eles estão certos.

III.2.3 DISCUSSÕES SOBRE O PRINCÍPIO DE DUALIDADE

LAUER e NEEDHAM reconhecem que não é muito fácil mudar a estrutura de vários sistemas operacionais, de maneira a refletir a dualidade sugerida. As estruturas de endereçamento subjacentes, o uso de dados globais e os estilos de comunicação são geralmente tão ligados ao projeto e à implementação, que realizar as transformações para uma versão dual seria um exercício tão custoso que não justificaria os ganhos de ordem secundário.

No entanto, pode ser citado o caso do computador CAP de Cambridge [117]. O sistema tem uma estrutura que permite um encapsulamento completo do endereçamento de cada módulo do sistema, e leva a um sistema operacional no qual cada módulo é implementado como um programa completo. Um princípio básico do projeto foi que qualquer estrutura de dados do sistema fosse gerenciada por um módulo único (ou procedimento protegido segundo a terminologia do CAP). Uma instância de tal procedimento pode ser encontrada em vários ou todos os processos do sistema. Esta estrutura facilita o tipo de reestruturação necessária para demonstrar a dualidade pretendida.

Em [160] STROUSTRUP discute os conceitos de LAUER e NEEDHAM que foram aplicados ao sistema operacional SIMOS (baseado no sistema CAP). Este foi simulado para um conjunto amplo de configurações de hardware e de cargas de programa. O sistema SIMOS foi escrito usando um conceito de módulo, que permite que um módulo individual possa ser interpretado como monitor numa execução e como processo em outra. As execuções que usam o monitor para controlar o acesso a alguns dados podem ser comparados com as execuções que usam um processo para controlar o acesso aos mesmos dados. Foram obtidos resultados iguais para o desempenho e o tempo de resposta calculados para os dois tipos de sistemas.

Se considerarmos que um serviço de um sistema pode ser implementado por um monitor ou por um processo, distinguiremos os dois estilos como o sistema baseado em monitor e o sistema baseado em processos. Se por outro lado nos concentramos na forma na qual são transmitidos os pedidos ao serviço, nos referiremos aos dois estilos de sistemas como sistema baseado em procedimentos e sistema baseado em mensagens.

STANTON em [155] e STROUSTRUP em [160] reforçam a idéia de que a distinção entre os dois tipos de sistemas é mais uma questão superficial de estilo do que uma questão de eficiência e de complexidade. Eles colocam que o problema principal na comparação entre os sistemas baseados em monitor e baseados em processo está nas duas palavras chaves processo e monitor que são utilizadas, cada uma, com significados bem diferentes. Por exemplo, cada sistema operacional implementa uma variação do conceito de processo. A hipótese da dualidade que pretende demonstrar a equivalência entre os dois estilos não pode ser comprovada em geral, embora possa ser aplicada a um conjunto particular de sistemas. É definido em [160] um domínio para o qual esta hipótese é válida e são também apresentados alguns exemplos de características que a violam.

Ainda em [155] é mostrada a grande utilidade do conceito de dualidade para a construção de programas concorrentes. A dualidade é considerada uma ferramenta que permite ao projetista de um programa concorrente escolher o estilo mais útil para facilitar sua compreensão e intuição, com a finalidade de escolher uma estrutura. A estrutura resultante, por dualidade, poderá ser transformada para o outro estilo para implementação. O autor ilustra estas idéias através da aplicação da ferramenta para alguns exemplos.

De acordo com estas idéias, GORDON afirma que durante um longo tempo acreditava que estender o núcleo do sistema operacional MUSS [165, 166], escrito no estilo baseado em troca de mensagens, utilizando o princípio de dualidade exigiria mudanças mínimas. No entanto ele enfatiza que na hora de implementar a conversão, surgiram problemas que levaram à conclusão de que o mapeamento se realiza facilmente na direção de sistemas baseados em troca de mensagens para os baseados em chamadas de procedimento mas não na direção contrária, a menos de restrições severas, bem mais fortes que as apontadas por LAUER e NEEDHAM.

Os problemas por ele levantados se referem ao caso de se ter chamadas aninhadas de monitores implementadas com liberação total da cadeia de chamadas no caso de acontecer algum bloqueio no nível mais interno, por exemplo por uma instrução do tipo WAIT. O exemplo que ele usa é o de, num sistema baseado em procedimentos,

um módulo A chamar um módulo B para executar algum serviço e esta chamada ser suspensa por um WAIT em B, com uma implementação que libere a exclusão dos monitores tanto de A quanto de B. Desta forma, A pode continuar atendendo pedidos de serviços, dos quais podem ser completados os que foram para operações diferentes. Isto contradiria a característica dos sistemas baseados em troca de mensagens colocada por LAUER e NEEDHAM, de que os processos operam sobre uma ou poucas mensagens de cada vez, e normalmente completam estas operações antes de consultar novamente a fila de mensagens.

GORDON afirma que as restrições severas necessárias para manter o princípio de dualidade não estão justificadas na base de uma boa prática de programação, e que são essenciais para o projetista de um sistema baseado em mensagens. Ele conclui que os dois estilos não são equivalentes e que considera mais poderoso o estilo baseado em chamada de procedimentos, já que só pode ser implementado o seu equivalente com um sub-conjunto de primitivas de troca de mensagens.

Em [58] GENTLEMAN, apesar de concordar com o princípio de dualidade dos dois modelos, aponta algumas diferenças interessantes. Ele considera que nos dois casos um programa de aplicação consistirá de algumas computações sequenciais e de uma série de transações com recursos compartilhados, onde existem interações potenciais com outros processos.

No entanto, o modelo orientado a procedimentos procura esconder a existência de outros processos e de suas interações com eles. O autor chama este modelo de **orientado a cliente**. O programa aparenta fazer tudo sozinho; acompanhando as chamadas de procedimentos pode ser visto todo o código necessário pelas transações do programa. Devido a estas características existe o risco de não reconhecer uma seção crítica, e mais ainda de não achar uma forma fácil de organizar o trabalho em recursos compartilhados, para otimizar a eficiência e o paralelismo. Por outro lado a associação clara de cada transação com um processo facilita a depuração e a recuperação depois das falhas.

É apontado que o modelo baseado em troca de mensagens procura esconder a identidade dos processos individuais que atendem ou provêem serviços. Este modelo é chamado pelo autor de **orientado a servidor**. Os processos servidores são projetados com tarefas bem

definidas e dependem de outros processos servidores para realizar outras tarefas. O programa é basicamente orientado a dados no nível dos processos. A prova de que uma computação individual de um programa de aplicação é completada, resulta do fato que cada servidor envolvido garante o término do serviço requerido por cada ítem da sua fila de trabalho. A exclusão mútua, a sincronização, o paralelismo e o balanceamento de carga decorrem naturalmente das características deste modelo. A sua desvantagem principal é devida a falta de associação da aplicação com a transação, que complica a recuperação das operações depois de alguma falha.

A opinião do autor é que a interface de um só nível do modelo servidor é mais modular, e portanto mais fácil de entender do que a interface de vários níveis imposta pelo modelo de cliente.

Fazendo uma analogia com a organização de pessoas numa empresa, ele define a sua preferência pelo modelo baseado na troca hierárquica de mensagens, isto é, pelo modelo orientado a servidor, que está mais próximo da intuição.

Foram levantadas em [155] as diferenças entre o modelo de troca de mensagens definido por GENTLEMAN e o de LAUER e NEEDHAM, e mostrado como, apesar disto, continua sendo possível aplicar o princípio de dualidade, contestado em parte por GENTLEMAN.

Estas críticas ilustram o fato ressaltado por LAUER e NEEDHAM de que a dualidade por eles apontada não pode ser demonstrada de maneira geral e que só existe para um domínio restrito de casos.

III.2.4 EXTENSÃO DO PRINCÍPIO DE DUALIDADE PARA SISTEMAS DISTRIBUIDOS

Como mencionamos no início da discussão sobre o princípio de dualidade, estas idéias podem ser aplicadas por extensão a programas distribuidos fisicamente sobre um conjunto de computadores. No caso de um sistema escrito no estilo baseado em troca de mensagens, a extensão é quase natural. Mas no caso de um sistema escrito no estilo baseado em chamada de procedimentos, ou em monitores, a extensão pode ser feita localizando o monitor que encapsula determinados dados e implementa um conjunto de

operações sobre esses dados, num específico nó físico. Os módulos que contenham chamadas para essas operações do monitor, no caso de estar carregadas em nós físicos diferentes do nó do monitor, precisam fazer chamadas remotas de procedimentos e não mais chamadas locais. A dualidade entre os dois estilos em sistemas distribuídos está baseada então na correspondência entre as chamadas remotas de procedimentos do monitor com operações apropriadas de troca de mensagens.

Desde que foi publicado por LAUER e NEEDHAM o trabalho sobre a dualidade das estruturas dos sistemas operacionais, foram criadas várias linguagens de alto nível para programação concorrente, e existe uma tendência a querer enquadrar as linguagens em duas categorias, as linguagens baseadas em procedimentos e as baseadas em mensagens. SCOTT discute esta classificação em [140], considerando-a enganosa, por confundir as características a ela associadas que ele considera independentes e que não deveriam ser definidas pelo projetista da linguagem.

Segundo ele, infelizmente, existem pelo menos duas perguntas que precisam ser respondidas para determinar a categoria à qual pertence a linguagem:

i) O emissor de uma mensagem continua a sua execução imediatamente ou espera por uma resposta?

ii) As mensagens são recebidas explicitamente por processos ativos ou a sua chegada provoca automaticamente a execução de algum código específico?

A primeira pergunta determina a escolha de um mecanismo de sincronização, enquanto a segunda está relacionada com a sintaxe da linguagem. O autor pretende mostrar que estas duas características são independentes e que as perguntas permitem formar quatro combinações de respostas e não duas. Idéias análogas a estas são expressas por ANDREWS e SCHNEIDER em [5], quando eles afirmam que a forma de nomeação e a sincronização são características ortogonais das primitivas de comunicação e sincronização.

Existem aplicações para as quais cada uma das quatro combinações será essencialmente apropriada e portanto, segundo SCOTT uma boa linguagem deveria prover mais de uma forma de combinação.

SCOTT faz uma análise exaustiva, no ambiente de sistemas distribuídos, das primitivas de sincronização utilizadas na literatura, mostrando que, segundo a resposta encontrada para a primeira pergunta, somente o envio remoto definido por LISKOV em [94] e a chamada remota de procedimento determinam que uma linguagem seja caracterizada como baseada em procedimentos. Dentro deste contexto DP (BRINCH HANSEN [29]), ADA(ICHBIAH et alii [73]), Communication Port (MAO e YEH [108]) e as outras linguagens baseadas no uso de monitores são do tipo baseado em procedimentos. Por outro lado, linguagens tais como PLITS (FELDMAN [55]) e Extended CLU (LISKOV [94]) assim como outras que usam troca de mensagens são do tipo baseado em mensagens.

O autor faz uma análise análoga sobre a forma de recepção de mensagens. Considerando as diferentes sintaxes de operações explícitas de recepção, e as formas implícitas que apresentam várias linguagens que usam o conceito de monitor, ele conclui que as linguagens que possuem recepção explícita pertencem à categoria baseada em troca de mensagens, e as que possuem recepção implícita pertencem à categoria baseada em procedimentos. Com esta classificação, em particular as linguagens Communication Port e ADA são agora linguagens baseadas em troca de mensagens. É interessante notar que na análise da primeira pergunta elas tinham sido caracterizadas como baseadas em procedimentos.

Algumas linguagens são nitidamente classificadas usando a dicotomia de baseadas em procedimentos ou baseadas em mensagens, independentemente de basear a classificação no mecanismo de sincronização ou na forma de recepção de mensagens. Mas, como já foi colocado no caso de ADA e Communication Port, esta classificação é ambígua e pode levar a resultados divergentes. Existe ainda o caso da linguagem *Mod (COOK [42]), que provê as quatro combinações possíveis: ambos tipos de sincronização combinados com recepção explícita e implícita. Isto demonstra que as características de sincronização e recepção de mensagens são totalmente independentes. Esta independência é colocada pelo autor como um argumento forte para questionar a validade da dicotomia de linguagens baseadas em mensagens e as baseadas em procedimentos.

Dependendo das aplicações, existem algumas combinações de

mecanismos de sincronização e de recepção de mensagens mais apropriadas que outras, e não existe uma única combinação que seja a mais adequada para todas as aplicações, como já foi mencionado anteriormente. O autor considera que, no caso de um sevidor que atende pedidos de uma comunidade de clientes, é mais apropriada a recepção implícita de mensagens. No entanto, no caso de uma interação entre módulos do tipo produtor/consumidor, a recepção explícita de mensagens se adequa melhor. Analogamente para os mecanismos de sincronização, não existe nenhuma forma única que satisfaça todos os tipos de aplicações. Portanto o autor conclui afirmando que assim como nas linguagens sequenciais é benéfica a presença de várias construções similares de laços, as linguagens concorrentes se beneficiam com a presença de vários mecanismos similares de comunicação entre processos.

Por outro lado HAMILTON em [64] discute também a extensão do conceito de dualidade para sistemas distribuidos, mas chegando a conclusões diferentes das apresentadas acima. Ele reconhece a equivalência do uso de troca de mensagens entre máquinas diferentes e a chamada remota de procedimento para a construção de sistemas distribuidos, e acha que a escolha por um determinado modelo é em parte arbitrária. Ele considera que mesmo sendo possível o suporte dos dois mecanismos por uma única linguagem, isto serviria somente para complicar a tarefa do implementador.

A troca de mensagens pareceria ser a alternativa mais flexível, já que uma única mensagem pode resultar num resultado de nenhuma, uma ou várias respostas. Por exemplo, um sistema de arquivos distribuido poderia estar organizado de maneira que todos seus servidores de arquivos enviassem os pedidos de transações automaticamente para o servidor do arquivo onde o arquivo está localizado. Neste caso o pedido seria enviado a um servidor de arquivo e a resposta voltaria a partir de outro servidor de arquivo. Na prática, entretanto, esta flexibilidade é raramente utilizada, e o que se encontra mais comunmente é um intercâmbio de envio e recepção de mensagens entre dois processos.

Se o protocolo de troca de mensagens pudesse evitar as mensagens de reconhecimento ("acknowledgment") para as mensagens recebidas (deixando a deteção e recuperação de erros para um nível mais alto de software), este modelo seria mais eficiente

que a RPC para tratar os casos nos quais não é necessário mensagens de reconhecimento individuais.

Em favor de RPC, pode-se observar que como a maioria das linguagens é essencialmente procedimental o uso de interfaces procedimentais para a comunicação remota evita uma mudança conceitual desnecessária. Esta característica das linguagens determina que as rotinas de biblioteca de baixo nível envolvidas nas interações remotas tendem a oferecer interfaces procedimentais para o software de nível mais alto.

HAMILTON aponta como exemplos para seus argumentos a tendência existente, tanto no sistema Tripos [132] quanto no sistema do anel de Cambridge [118], da troca de mensagens se transformar em intercâmbios da forma pedido/resposta, confirmando a hipótese de que o modelo procedimental é mais satisfatório.

CAPÍTULO IV

MODELO A SER ACRESCENTADO A MODULA-2 PARA PROGRAMAÇÃO EM PEQUENA ESCALA

Nos dois capítulos anteriores, analisamos os sistemas distribuídos, as suas características, a sua estruturação modular e os requerimentos de linguagens para a sua programação. Analisamos também os mecanismos de comunicação e sincronização entre processos para a programação em pequena escala dos sistemas distribuídos. Neste capítulo apresentaremos uma proposta em relação a este último aspecto para a construção do ambiente de programação distribuída, baseado no uso da linguagem Modula-2.

IV.1 JUSTIFICATIVAS

IV.1.1 ANÁLISE DE OUTROS MODELOS PARA SISTEMAS DISTRIBUÍDOS

Achamos na literatura duas filosofias diferentes para a construção de ambientes distribuídos, uma baseada na definição e projeto de uma nova linguagem e outra que estende alguma linguagem já existente. Analisaremos a seguir um exemplo de cada uma destas filosofias.

ANDREWS et alii [8] estavam projetando o sistema operacional distribuído SAGUARO que consiste de um conjunto de computadores ligados por uma rede local. O sistema está sendo desenvolvido na linguagem SR (ANDREWS [4, 6, 7]) definida com esta finalidade, e que justamente por isto tem sofrido uma série de modificações, desde a sua primeira definição [4], à medida que surgiram novas necessidades. Uma das principais mudanças foi a de tornar os elementos básicos da linguagem (recursos e processos) dinâmicos em vez de estáticos. A outra mudança é que as operações e os processos estão agora integrados numa nova forma: todos os mecanismos de interação entre processos (chamadas locais e remotas, "rendezvous", criação dinâmica de processos e troca de mensagem assíncrona) são expressos de forma sintaticamente similar.

Em contraposição a este enfoque, as idéias que nortearam o nosso projeto foram inspiradas, em parte, no projeto Cnet [40], e em particular no sub-projeto intitulado "Sistemas Distribuídos para Redes Locais". Este projeto, promovido pelo "Consiglio

Nazionale delle Ricerche, Progetto Finalizzato Informatica", foi desenvolvido na Italia durante o período de 1980 a 1985, por dezessete grupos de pesquisadores de várias regiões da Italia, pertencentes à indústria, centros de pesquisa e universidades. O seu objetivo inicial era definir as especificações funcionais de uma arquitetura distribuida baseada numa rede local, e tomando como estudo de caso a automação de escritórios.

Queremos destacar aqui a filosofia utilizada pelo projeto Cnet no qual foi adotada a linguagem ADA, tanto de linguagem de sistema, como de linguagem para as aplicações num sistema distribuido. Esta escolha responde a um dos objetivos do projeto, que consiste em mostrar a importância do uso de uma linguagem de alto nível em programação distribuida. De fato, apesar de Cnet preservar a autonomia dos nós individuais carregados em estações de trabalho hospedeiras heterogêneas, o uso da linguagem de alto nível garante a percepção de um único sistema distribuido coerente. Em Cnet é feita a distinção entre a visão lógica ou virtual do sistema, e a visão do hardware. Somente a primeira é oferecida aos projetistas do sistema e das aplicações. Mesmo assim os projetistas podem referenciar e controlar os recursos disponíveis, definindo estratégias diferentes através da linguagem de alto nível que incorpora primitivas específicas dos sistemas operacionais. Estas facilidades são típicas de um enfoque integrado [97, 94, 154].

Com este objetivo foram acrescentados, como extensão da linguagem Ada, mecanismos abstratos para implementar: a comunicação entre nós, o tratamento de tolerância a falhas e a configuração do sistema.

Neste capítulo trataremos somente da comunicação entre nós e analisaremos as outras extensões em capítulos seguintes.

O modelo escolhido pelo projeto Cnet implementa comunicação assíncrona com controle descentralizado. Ele provê mensagens e portas com tipo para possibilitar verificação de maneira estática, criação e ligação dinâmica de portas, e modularidade. Um dos pontos mais importantes a destacar é que foi mantida, no ambiente distribuido, a definição de programa em Ada, de tal maneira que os programas distribuidos devem ser definidos como uma única biblioteca de programas.

A noção de biblioteca de programas Ada, aplicada ao ambiente

distribuido, é essencial para descrevê-los como uma coleção de nós. O nó (NODE) é a entidade lógica de nó virtual; ele é composto de uma coleção de unidades da biblioteca padrão ADA que interagem e se comunicam usando os mecanismos padrões da linguagem. Por outro lado somente a comunicação indireta por troca de mensagens é permitida entre nós diferentes, isto é, entre tarefas pertencentes a nós diferentes.

Em tempo de execução podem ser distinguidos dois níveis de visibilidade do sistema Cnet:

- no nível do sistema distribuido, a execução do nó (NODE) é a unidade elementar de computação;

- no nível do nó (NODE) a unidade elementar de computação é a tarefa.

Devido à ausência de memória comum, os nós compartilham um ambiente global no qual as unidades de biblioteca são unicamente unidades genéricas e pacotes, dos quais a parte declarativa contém somente declarações de tipo (exceto declarações de tipo acesso) e declarações de unidades genéricas, e o corpo ("body") não contém comandos.

A biblioteca de programas da rede é definida usando as facilidades de configuração disponíveis no ambiente Cnet, que permitem projetar um nó (NODE) da forma anteriormente mencionada. Esta biblioteca inclui pacotes genéricos de comunicação para implementar três tipos de portas (difusão, entrada e saída) e suas operações, e estabelecer o ambiente global no qual uma aplicação distribuida de ADA pode ser projetada.

IV.1.2 APRESENTAÇÃO SUCINTA DA NOSSA ESCOLHA

Na introdução foram apresentados os argumentos que justificam a nossa decisão de escolher uma linguagem de alto nível já divulgada e aceita internacionalmente, em vez de definir uma nova linguagem para desenvolver o ambiente de programação distribuida.

Entre as novas linguagens de programação que apareceram no final da última década, Modula-2 (WIRTH [178, 180], SEGRE e STANTON [144]) acrescenta às características de Pascal facilidades para o desenvolvimento modular (que separam a

definição da interface de um módulo da sua implementação), compilação separada, programação concorrente e criação de tipos abstratos. Além disto, Modula-2 permite o manuseio direto dos mecanismos de endereçamento de memória, de interrupções e de entrada/saída em sistemas monoprocessador. Com estas características torna-se possível escrever todo o software de um sistema de computação nesta linguagem, como ilustrado no caso do sistema MEDOS-2 para a máquina Lilith [179].

Um aspecto importante desta linguagem é sua disponibilidade para alguns minicomputadores (PDP-11, VAX-11), e em particular para vários microcomputadores, inclusive o Apple II, o PC da IBM e o Macintosh.

Além das vantagens de Modula-2, enumeradas na introdução, queremos salientar que ela satisfaz a maioria dos requisitos das linguagens para sistemas distribuídos, analisados no Capítulo II, em particular, para a programação em pequena escala.

À Modula-2, escolhida para este trabalho, vão ser acrescentadas extensões suficientes para atender às necessidades de software distribuído. Em particular, neste capítulo apresentaremos as extensões necessárias para a implementação do conceito de processo e para os mecanismos de comunicação e sincronização entre processos.

Estas extensões serão implementadas através de módulos que exportam os conceitos desejados, de maneira que os sistemas possam ser programados usando Modula-2, sem modificar seu compilador. Esses conceitos serão importados dos módulos acrescentados, de forma análoga à apresentada no projeto Cnet.

Módula-2 oferece o conceito de co-rotina, que pode ser utilizado para implementar processos concorrentes num monoprocessador, através de um núcleo, ou seja, um módulo que exporta o conceito de processo [143]. Num sistema distribuído, cada estação suportará a execução concorrente de vários processos, através de um núcleo próprio.

Para os mecanismos de comunicação e sincronização entre processos, consideramos em maior sintonia com o espírito de Modula-2 a escolha do modelo de comunicação entre processos baseado em procedimentos (isto é, através de monitores), ao invés da alternativa de troca de mensagens (discussão de LAUER e NEEDHAM [90]). Esta escolha tem como consequência a necessidade

de implementar chamadas remotas de procedimentos como principal mecanismo de comunicação entre estações. As extensões necessárias para realizar estas idéias incluem a implementação no núcleo de programação concorrente das operações de sincronização utilizadas dentro de monitores, e o uso de "stubs" (BIRRELL e NELSON [19]) para implementação de chamada remota de procedimento.

O modelo de comunicação e sincronização entre processos adotado por Cnet mantém o "rendezvous" de Ada para processos dentro de um mesmo nó, e acrescenta a troca de mensagens através de portas para a comunicação entre processos de nós diferentes. No nosso caso adotamos um único mecanismo de comunicação entre processos. A comunicação através de chamada de procedimento, seja ela local ou remota, dependendo se os processos estão no mesmo nó físico ou não, permite ao programador desconhecer a configuração na qual o sistema será executado. Para isto é necessário implementar um modelo de chamada remota de procedimento (RPC) que seja transparente para o usuário. Este modelo, que foi implementado inicialmente por NELSON [119] e BIRRELL e NELSON [19] para ser acrescentado à linguagem MESA (MITCHELL et alii [113]) norteou nosso trabalho. A implementação desenvolvida para a utilização de Modula-2 na programação em ambientes distribuidos será apresentada nas próximas seções deste capítulo.

Em relação à escolha de RPC, podemos observar que a maioria das linguagens são fundamentalmente procedimentais, e o uso de interfaces procedimentais para comunicação remota evita uma mudança conceitual desnecessária. O desenvolvimento de RPC é uma evolução natural dos conceitos de abstração de dados, controle e concorrência, das linguagens procedimentais. Esta evolução pode ser ilustrada resumidamente da seguinte forma: às noções básicas de procedimentos de Algol (NAUR [116]) foram acrescentadas as especificações de tipo em Pascal (WIRTH [172]). BRINCH HANSEN estendeu Pascal com monitores e processos em Pascal Concorrente [27]. Como método de abstração, foram acrescentados módulos em MESA (MITCHELL et alii [113]) e Modula (WIRTH [176]). Finalmente em MESA e ADA (ICHBIAH [73]) foram implementadas a compilação separada de módulos e monitores com controle rígido de tipos.

Como foi colocado por LISKOV et alii em [100], a ênfase procedimental das linguagens existentes faz com que as rotinas de

biblioteca de baixo nível, envolvidas nas interações remotas, tendam a oferecer interfaces procedimentais ao software de mais alto nível. RPC representa de modo natural este tipo de interface.

IV.1.3 USO DE RPC EM DIFERENTES NÍVEIS E FORMAS

O uso de RPC pode ser encontrado na literatura em dois níveis diferentes, no nível da linguagem e no nível de protocolo de transmissão da rede.

No primeiro nível, a chamada remota de procedimento é um mecanismo que permite que uma chamada feita a nível da linguagem numa máquina, cause automaticamente a chamada correspondente na outra máquina. Tal mecanismo precisa de um protocolo de rede para dar suporte à transferência de seus argumentos e resultados. Ás vezes o uso de RPC é utilizado para descrever este nível de protocolo de transferência.

Citaremos nesta seção vários exemplos encontrados na literatura sobre o uso cada vez mais difundido deste mecanismo, nas suas diferentes formas, mas aqui queremos chamar atenção para o fato de que ele está sendo considerado o mais adequado para distintos modelos de sistemas distribuídos. Vários grupos de pesquisa defendem a adequação do uso de RPC, especialmente para sistemas distribuídos baseados no modelo cliente/servidor, mas podemos encontrar vários projetos de outros tipos adotando este mecanismo também. Vale ainda ressaltar que numa conferência promovida pela ACM SIGOPS [121] em dezembro de 1985 sobre "Accommodating Heterogeneity", foi unânime o consenso do uso de RPC para sistemas heterogêneos, mesmo levando em consideração alguns problemas associados ao seu uso e em particular à sua semântica.

Em relação aos dois níveis aos quais nos referimos antes, um exemplo de uso de RPC no nível mais baixo é o projeto denominado "Newcastle Connection" [147, 161, 129], no qual o mecanismo RPC não está integrado à linguagem, e somente dá suporte a algumas rotinas padrão que codificam e decodificam os argumentos e os resultados a partir de áreas de memória ("buffers") da rede.

Ao nível da linguagem podemos considerar duas filosofias

diferentes de implementação de RPC, uma que oferece o mecanismo embutindo-o na própria linguagem, e outra que oferece o mecanismo, ou através de uma outra linguagem da rede, ou através de extensões que podem ser chamadas pelo software da rede. Como exemplos de sistemas que usam este último enfoque podemos relacionar os seguintes:

i) O projeto Apollo DOMAIN [92, 161, 129], que é uma arquitetura para redes de estações de trabalho pessoais e servidores, que cria um ambiente integrado de computação distribuida. Inicialmente o objetivo do projeto era implementar um sistema de arquivos homogêneo e distribuido, porém mais recentemente o sistema foi ampliado para atender a um ambiente heterogêneo. Desde 1985 [129] está sendo desenvolvido um mecanismo de RPC a ser utilizado acima do protocolo básico de serviço da rede para a interface entre os clientes e os servidores.

ii) O projeto ADMIRAL [10, 129] que desenvolve um sistema de estações de trabalho para um ambiente heterogêneo utiliza RPC para a comunicação entre as estações. O modelo de RPC é baseado no de NELSON [119] mas é independente da linguagem de pequena escala. Para isto é definida uma nova linguagem Glue (BACARISSE [11]), na qual são especificadas as interfaces dos serviços oferecidos aos clientes, que serão traduzidas para a linguagem de implementação automaticamente.

iii) Outro projeto semelhante ao anterior é o desenvolvido no System Research Center da Digital Equipment Corporation (DEC) [21] que liga estações de trabalho através de uma rede local. O objetivo do projeto é poder executar programas aplicativos já desenvolvidos para o sistema operacional UNIX sem precisar modificá-los. É utilizado o mecanismo de RPC baseado em NELSON para a comunicação entre espaços de endereços e máquinas distintos. Foi implementado um programa que aceita interfaces escritas em Modula-2 e gera módulos "stubs" que serão utilizados pelo sistema de suporte em tempo de execução através de chamadas ao núcleo de UNIX.

iv) Em [129] é apresentado outro projeto de uma arquitetura

distribuida que está sendo desenvolvido no DEC por EMER e RAMAKRISHNAN, e cujo objetivo é abordar a heterogeneidade, tanto no uso de máquinas diferentes quanto no uso de vários sistemas operacionais. O sistema distribuido está baseado no modelo cliente/servidor, no qual um determinado número de serviços centralizados do sistema está disponível de forma remota aos clientes dos nós da rede local. No nível abaixo do sistema de interfaces de serviços, é utilizado um mecanismo de acesso que consiste de uma linguagem de comunicação entre clientes e servidores. A linguagem implementa RPC acima do protocolo pedido/reposta.

v) Como último exemplo podemos mencionar o projeto EDEN que está em vias de desenvolvimento na Universidade de Washington [17], que implementa um mecanismo interessante de RPC para um ambiente heterogêneo. Os autores provêem interfaces procedimentais suficientemente poderosas para escrever aplicações distribuidas que usam uma RPC heterogênea (HRPC) para ter acesso, ou ser utilizada por três outros mecanismos de RPC existentes. Os "stubs" do componente de HRPC são adaptados de modo a complementar os componentes de RPC dos outros estilos em tempo de execução, aparentemente sem perda de desempenho. Os mecanismos de RPC, com os quais a HRPC se interliga, se apresentam todos com a mesma semântica e são similares. Isto representa um passo a frente no caminho de definir uma RPC padrão com custos baixos e aceitáveis.

Poderíamos citar outros trabalhos (BLOOM [24], BOCHMANN e RAYNAL [25]), mas como o nosso interesse está particularmente voltado para o caso de uso de RPC no nível da linguagem e nela embutido, passaremos a citar alguns exemplos que seguem esta filosofia para depois tratá-los com mais profundidade em outra seção deste capítulo.

A primeira proposta de uso de RPC embutida numa linguagem foi a desenvolvida por NELSON [119] e implementada depois por BIRRELL e NELSON [19] na linguagem MESA (MITCHELL et alii [113]), e será tratada com mais detalhe posteriormente. O mecanismo por eles proposto é transparente, isto é para o programador não existe diferença entre uma chamada de procedimento local ou

remota, ou seja, a sintaxe da chamada é a mesma, independente da localização física do procedimento que está sendo chamado.

Em contraposição a esta proposta, HAMILTON [100] apresentou posteriormente um mecanismo de RPC explícito que foi embutido numa versão estendida da linguagem CLU (LISKOV e SCHEIFLER [97]). Ele considera que não pode ser ignorada a enorme diferença de desempenho que existe entre chamadas locais e remotas, e as diferenças na semântica em relação à possibilidade de falhas entre as duas. Compararemos estas duas propostas com mais detalhe mais adiante.

Seguindo o modelo de NELSON [119], isto é, no estilo de RPC transparente, podemos mencionar a linguagem SR (ANDREWS [4]) e uma extensão de "Modular Pascal" (BRON [31]) proposta por ROSSEM [136]. No projeto SAGUARO [8], que usa a linguagem SR, está sendo proposto estender a implementação de RPC, para ser utilizada por diferentes linguagens, aproveitando os programas já escritos em outras linguagens.

Outro exemplo que podemos citar é o da linguagem ARGUS (LISKOV [99], LISKOV et alii [98]), que originalmente implementava um sistema de comunicação baseado na troca de mensagens, mas que foi alterada posteriormente por seus projetistas para o esquema procedimental, por o considerarem mais confiável para a construção de software distribuido [147]. ARGUS, que também é baseada numa linguagem já existente (CLU (LISKOV et alii [95])), oferece tratamento de exceções em RPC através de uso de réplicas para exceções transitórias, e reconfiguração para falhas.

Recentemente foi divulgada uma extensão de Modula-2 para sistemas distribuidos homogêneos e fortemente acoplados (MELLOR et alii [111]). As extensões incluem chamada remota de procedimentos para a comunicação entre processadores e existem ligações de dois tipos, relativas e fixas, que são indicadas na linguagem de programação. A alocação do código de programa e dos dados é feita de forma independente da linguagem permitindo realocação sem precisar recompilar. Serão analisadas no Capítulo VII as diferenças entre esta implementação e a apresentada neste trabalho.

O modelo escolhido, como mecanismo de comunicação e sincronização entre processos a ser acrescentado à Modula-2, foi

o de RPC embutido na linguagem e transparente, de maneira que os sistemas distribuídos possam ser programados independentemente da configuração na qual o sistema será executado. A configuração é definida separadamente com o uso de uma linguagem especialmente projetada que será apresentada no Capítulo VI.

Apresentaremos a seguir os diferentes núcleos de multiprogramação que foram implementados para exportar o conceito de processo, e depois de uma discussão mais detalhada sobre as diferentes características de RPC no nível da linguagem, será descrita a nossa implementação.

IV.2 NÚCLEOS DE MULTIPROGRAMAÇÃO

IV.2.1 INTRODUÇÃO

Os programadores que trabalham em grandes projetos de software, como por exemplo sistemas operacionais, têm sentido a necessidade de dispor de uma linguagem que combine as melhores características de programação estruturada junto com a possibilidade de ter acesso diretamente ao "hardware" subjacente.

As linguagens mais recentes para multiprogramação, tais como Modula (WIRTH [176]), Pascal Concorrente (BRINCH HANSEN [27]), Pearl (DIN [49]) e Ada (ICHBIAH et alii [73]), incluem um método fixo para representar atividades concorrentes (processos, tarefas), para comunicação entre elas (semáforos, sinais, mensagens, "rendezvous") e para tratamento de entrada/saída.

Os sistemas escritos nestas linguagens estão baseados num programa residente chamado normalmente núcleo ("kernel"), que implementa processos, comunicação, sincronização e E/S. Este núcleo é geralmente a parte mais crucial do sistema operacional e ele determina a sua eficiência, confiabilidade e desempenho.

A tarefa de escrever o núcleo é difícil devido à complexidade dos conceitos implementados por ele, ao seu comportamento dependente do tempo, e à insuficiência de ferramentas de programação usadas para a sua implementação. A maioria dos núcleos é escrita em linguagem de montagem, que não oferece nenhum tipo de programação estruturada, complicando a sua codificação e capacidade de detectar erros dependentes do tempo, além de dificultar a manutenção.

As linguagens baseadas num núcleo permitem um acesso parcial

ao hardware do computador. As primitivas de sincronização, as interrupções e os registradores associados aos dispositivos periféricos estão escondidos do programador. O acesso a estas características é permitido somente através de chamadas às primitivas do núcleo.

Se por um lado isto oferece a vantagem para o programador de dispor de uma máquina virtual mais poderosa, sem ter que se preocupar com detalhes mais ligados ao hardware da máquina, por outro lado, para projetos de software que precisam de um maior controle sobre a execução dos processos, esta filosofia implementa políticas para gerenciamento de dispositivos, escalonamento de processos e tratamento de interrupções fixas que não necessariamente são as mais adequadas para determinadas aplicações.

Modula-2 (WIRTH [180]), uma linguagem para programação de sistemas, oferece a possibilidade de escrever grandes sistemas de software em todos seus níveis, eliminando totalmente a necessidade de programar em linguagem de montagem os níveis mais baixos do software. Ela é uma ferramenta poderosa para construir software desde o hardware até a interface homem-máquina.

Modula-2 não oferece certas das características disponíveis em outras linguagens, tais como procedimentos predefinidos para entrada/saída, alocação de memória e escalonamento de processos, mas ela possui as ferramentas básicas necessárias para programar estas características, de forma que o programador possa escolher os mecanismos e as políticas mais apropriadas em cada caso.

Em Modula-2 o nível de abstração destas facilidades é bastante próximo ao do nível do hardware subjacente, permitindo uma relativa eficiência de programação, o que não ocorre sempre quando uma abstração de mais alto nível é implementada. É conveniente impor uma estrutura mais complexa em cima dos mecanismos primitivos de co-rotinas e interrupções de Modula-2, para poderem ser usados com facilidade e segurança para uma determinada aplicação. Isto pode ser feito definindo módulos que oferecem abstrações de mais alto nível, como por exemplo, módulos equivalentes a núcleos que utilizem diferentes filosofias para comunicação e sincronização de processos, e módulos correspondentes a gerentes de dispositivos que controlem o tratamento de interrupções.

A estrutura modular de Modula-2 permite definir uma hierarquia de abstrações de máquinas virtuais partindo do nível mais baixo, que é o núcleo acima mencionado. Será mostrado neste trabalho também, esta hierarquização de módulos para a implementação de processos concorrentes.

Para ilustrar estas idéias, foi desenvolvido em [143] o conhecido exemplo de produtores e consumidores, analisando o caso de ter um processo produtor e um processo consumidor que se comunicam e se sincronizam trocando informações, seja através de memória compartilhada, ou seja através de troca de mensagens. Foram analisados os dois casos por separado, mostrando como as ferramentas de Modula-2 permitem escrever núcleos que implementem as duas filosofias de comunicação e sincronização.

Para ilustrar o uso das ferramentas para tratamento de interrupções foi tratado o exemplo de um módulo que implementa o compartilhamento de processador entre os diferentes processos atribuindo uma fatia de tempo constante para cada um.

IV.2.2 CARACTERÍSTICAS PARA IMPLEMENTAÇÃO DE PROCESSOS

IV.2.2.1 MECANISMOS OFERECIDOS PELA LINGUAGEM

Modula-2 (WIRTH [180]) foi projetada inicialmente para ser implementada num computador convencional de um único processador. Ela oferece algumas facilidades básicas para multiprogramação que permitem a especificação de processos quasi-concorrentes e de real concorrência no caso de dispositivos periféricos como será visto mais adiante.

A palavra processo é aqui utilizada com o significado de co-rotina. Uma co-rotina é um programa sequencial, com suas próprias variáveis particulares, que é executada em regime de quase concorrência junto com outras co-rotinas num único processador. O processador é comutado de uma co-rotina para outra através de um comando de transferência explícita de controle. A co-rotina que recebe controle retoma sua execução a partir do ponto da sua última suspensão através de comando de transferência de controle.

Como em Modula-2 co-rotinas são consideradas facilidades de baixo nível, o tipo associado e seus operadores fazem parte do pseudo módulo chamado SYSTEM. Para que outros módulos possam usar estas ferramentas eles devem importá-las a partir de SYSTEM.

Os programas referenciam as co-rotinas através de variáveis de tipo PROCESS que podem ser consideradas como apontadores para a co-rotina. Para poder distinguir as co-rotinas, uma variável de tipo PROCESS deve ser declarada para cada co-rotina a ser criada.

Uma co-rotina é criada pela chamada do procedimento NEWPROCESS do pseudo módulo SYSTEM:

```
PROCEDURE NEWPROCESS(Procedimento:PROC;EnderecoAreaTrabalho:
            ADDRESS;TamanhoAreaTrabalho:CARDINAL;
            VAR NovoProcesso: PROCESS);
```

Uma vez criada, a co-rotina pode ser executada, mas ela permanece inativa até que outra co-rotina lhe transfira o controle.

As co-rotinas alternam entre dois estados - inativo e executando - com somente uma executando de cada vez. As transferências de co-rotinas se parecem mais com comutação de processos, mas em lugar de ser gerenciadas por um escalonador de processos separado, a co-rotina em execução, indica explicitamente qual será a próxima a ser executada.

A transferência de controle entre duas co-rotinas é especificada pela chamada do procedimento TRANSFER do pseudo módulo SYSTEM:

```
PROCEDURE TRANSFER (VAR fonte, destino: PROCESS);
```

o efeito desta chamada é suspender a co-rotina em execução, atribuir seu apontador ao parâmetro fonte e reativar a co-rotina apontada pelo parâmetro destino, a partir do último ponto onde ela tinha sido suspensa.

Utilizando o conceito de co-rotina, é possível agora construir módulos que implementam o conceito de processo concorrente, com diversos modelos tanto para a sua definição quanto para as suas interações, e com diferentes opções de políticas de escalonamento. Na prática, isto é feito através da programação de um módulo que exporta os conceitos apropriados para o usuário (programador de software concorrente). Usando a divisão entre módulo de definição e módulo de implementação [76], os detalhes da implementação destes núcleos podem ser totalmente escondidos do usuário.

O grupo de WIRTH já publicou dois núcleos para programação

concorrente escritos em Modula-2. WIRTH [180] descreveu um módulo chamado "Processes" que implementa um administrador de processsos, que se sincronizam através das primitivas "Wait" e "Send". Estas primitivas podem ser usadas para implementar os módulos de interface da linguagem Modula (WIRTH [176]), que são parecidos aos monitores de HOARE [67] e BRINCH HANSEN [27], permitindo que processos concorrentes tenham acesso a variáveis globais em regime de exclusão mútua. Em outro trabalho, HOPPE [72] apresentou um núcleo que implementa comunicação e sincronização através da troca de mensagens entre processos concorrentes por meio de caixas postais.

IV.2.2.2 MÓDULO NUCLEOHOLT

Escolhemos implementar o núcleo de HOLT et alii [70], que considera políticas diferentes de escalonamento e de comunicação e sincronização de processos, do modelo implementado por WIRTH, embora ele também forneça ferramentas para implementar o conceito de monitor.

O módulo programado para implementar o núcleo de HOLT, cuja listagem se encontra no Anexo 1 de [143], para poder ser utilizado nas diferentes partes de um programa escrito em Modula-2, precisa ser uma unidade de compilação. Portanto ele constará de uma parte de definição, representando a interface com os outros módulos, que contém os nomes dos procedimentos e tipos por ele implementados, que podem ser importados pelos outros módulos. A parte de implementação é a que contém os detalhes dos tipos e os algoritmos correspondentes aos procedimentos que podem ser exportados, além de variáveis, procedimentos e módulos locais e a inicialização do módulo.

Para programar o módulo, é necessário importar do módulo SYSTEM os tipos e as primitivas que estão relacionadas com o tratamento de processos, vistos na seção anterior. Precisamos também importar primitivas pertencentes ao módulo que gerencia a memória, para alocar e liberar áreas de memória.

Na nossa implementação cada processo tem um descritor de processo a ele associado, e é criado um tipo FilaProc que corresponde a uma fila de descritores de processos, cuja declaração está no módulo de implementação, para ser exportado de

forma opaca, com a seguinte estrutura ilustrada pela figura (IV.1):

```
TYPE FilaProc = POINTER TO DescritorProcesso;
DescritorProcesso =
  RECORD  contexto :  PROCESS;      (*área para salvar o contexto
                                     em caso de interrupção*)
          seguinte :  FilaProc;     (*ligação com o próximo
                                     descritor da fila*)
          basepilha : ADDRESS;      (*usado para poder*)
          tamanho : CARDINAL;       (*devolver a área da pilha*)
          prioridade : CARDINAL     (*só utilizado quando é
                                     implementada espera com
                                     prioridade*)
END;
```

FIGURA IV.1 - DESCRIÇÃO DO TIPO DescritorProcesso

Este tipo é utilizado para implementar as filas de condição, uma para cada condição, e a fila dos processos prontos cujo primeiro elemento é o processo que está executando.

Para descrever as primitivas implementadas pelo NucleoHolt e que podem ser exportadas junto com o tipo FilaProc, apresentaremos a programação do módulo de definição ilustrado pela figura (IV.2):

```
DEFINITION MODULE NucleoHolt;
  EXPORT  QUALIFIED  FilaProc,  CriaProcesso,  Espera,  Avisa,
          FilaVazia, Inicializa, FimProcesso;
  TYPE FilaProc;

  PROCEDURE CriaProcesso(CodigoProcesso:PROC;tamanhopilha:
      CARDINAL);
    (*cria um processo associado ao código do procedimento
     CodigoProcesso e com uma área de trabalho de tamanho
     tamanhopilha*)

  PROCEDURE Espera (VAR C:FilaProc);
    (*coloca o processo numa fila esperando que outro sinalize a
     condição C*)
    (*ou*)

  PROCEDURE EsperaP (VAR C:FilaProc; Pr:CARDINAL);
    (*coloca o processo numa fila ordenada pela prioridade Pr
     esperando que outro sinalize a condição C*)

  PROCEDURE Avisa (VAR C:FilaProc);
    (*ativa um processo que esteja esperando pela condição C*)

  PROCEDURE FilaVazia (C:FilaProc):BOOLEAN;
    (*informa se a fila de condição tem algum elemento*)

  PROCEDURE Inicializa (VAR C:FilaProc);
    (*inicializa a fila correspondente à condição C*)

  PROCEDURE FimProcesso;
    (*acaba a execução do processo e desaloca a área de trabalho
     dele*)

END NucleoHolt.
```

FIGURA IV.2 - MÓDULO DE DEFINIÇÃO DO NucleoHolt

A inicialização do módulo consiste na alocação de um descritor de processo e a sua inicialização.

No caso da primitiva Espera, HOLT definiu uma primitiva Wait com prioridade para a espera pela condição. Para poder implementar esta primitiva que foi chamada EsperaP, precisamos acrescentar ao descritor de processo um outro campo para armazenar o valor da prioridade. Neste caso a única primitiva que é alterada é a primitiva EsperaP que vai inserir o processo bloqueado na fila de condição na ordem estabelecida pela prioridade associada. Desta maneira a primitiva Avisa ativará sempre o processo com maior prioridade. No caso de ter vários processos com a mesma prioridade é usada a política FIFO para inserir o processo.

IV.2.2.3 MÓDULO DE PROGRAMA PRODUTOR CONSUMIDOR

O módulo NucleoHolt foi utilizado para implementar o programa chamado ProdutorConsumidor, que será um Program Module e cuja listagem é apresentada na Figura (IV.3). Ele contém como módulo local o módulo chamado Buffer, que implementa o conceito de monitor. Este é utilizado para a comunicação e sincronização dos dois processos, Produtor e Consumidor, sendo que um envia e o outro consome mensagens no Buffer.

No final da inicialização, se mata o processo inicial transferindo o controle para os processos criados, seus filhos.

Este exemplo pretende ilustrar como, uma vez programado o núcleo que implementa o conceito de processos escolhendo uma determinada política de gerenciamento, podem ser definidos níveis abstratos de maior nível hierárquico, que facilitam a programação de um sistema complexo: podemos destacar aqui três níveis: o núcleo, o buffer e o programa que contém os processos.

```
MODULE ProdutorConsumidor
  FROM InOut IMPORT Write, WriteInt, Read;
  FROM  NucleoHolt  IMPORT  CriaProcesso,   FimProcesso,  Espera,
          Avisa, FilaProc, Inicializa;
    CONST N=300   (*tamanho área de pilha*)

  MODULE Buffer [6];
    IMPORT Espera, Avisa, FilaProc, Inicializa;
    EXPORT Obter, Depositar;
      CONST N=128;  (*tamanho buffer*)
      VAR n:INTEGER;  (*no. de elementos depositados*)
      nvazio:FilaProc;    (*n<N*)
      ncheio:FilaProc;    (*n>0*)
      ent,said:[0..N-1];  (*indices*)
      buf:ARRAY [0..N-1] OF CHAR;

       PROCEDURE Depositar (X:CHAR);
         BEGIN n:=n+1;
         IF n>N THEN Espera(ncheio) END;
            (*n<N*)
         buf [ent]:=X; ent:=(ent+1)MOD N;
         IF n<=0 THEN Avisa (nvazio)END
       END Depositar;

       PROCEDURE Obter (VAR X:CHAR);
         BEGIN n:=n-1;
           IF n<0 THEN Espera(nvazio) END;
              (*n>=0*)
           X:=buf [said]; said:=(said+1)MOD N;
           IF n>=N THEN Avisa(ncheio) END
       END Obter;

     BEGIN n:=0; ent:=0; said:=0;
       Inicializa(ncheio); Inicializa(nvazio)
     END Buffer;

  PROCEDURE Produtor;
    VAR X:CHAR;
      LOOP
        Read(X);
        Depositar(X)
    END;    (*LOOP*)
  END Produtor;

  PROCEDURE Consumidor;
    VAR X:CHAR;
        I:CARDINAL;
    BEGIN
    I:=0;
     LOOP
       I:=I+1;
       Obter(X);
       WriteInt (I,4); Write(X)
     END;     (*LOOP*)
   END Consumidor;

BEGIN
  NucleoHolt . CriaProcesso (Produtor,N);
  NucleoHolt . CriaProcesso (Consumidor,N);
  FimProcesso
END ProdutorConsumidor.
```

FIGURA IV.3 - PROGRAMA DO ProdutorConsumidor (1a. VERSÃO)

IV.2.2.4 MÓDULO NucleoTrocaMensg

Mostramos agora outro exemplo de núcleo que implementa processos, no qual a comunicação e sincronização entre eles são realizadas através da troca de mensagens. As operações sobre

mensagens, que são atômicas (não interrompíveis), podem ser chamadas em qualquer ponto do programa e são implementadas por chamadas normais de procedimentos, para as quais o compilador testará o número e os tipos dos parâmetros.

A propriedade atômica dos procedimentos que implementam as operações é obtida, declarando-os dentro de um módulo com alta prioridade, de maneira que as interrupções sejam inibidas durante sua execução.

Em [72] é apresentado um exemplo de núcleo baseado em troca de mensagens no qual os processos se comunicam enviando e recebendo mensagens através de uma caixa postal que atua como elemento de sincronização. Como este exemplo está amarrado ao caso de arquitetura com memória compartilhada, quisemos projetar primitivas que tivessem como parâmetro os próprios processos envolvidos na comunicação, para depois poder estender este caso a sistemas distribuidos sem memória compartilhada. Para analisar estas primitivas vamos primeiro mostrar as estruturas de dados utilizadas no nosso núcleo.

Cada processo tem um descritor de processo associado com a estrutura mostrada a seguir. Os descritores de processos estão ligados em forma de anel, de maneira que cada um contém o ponteiro para o próximo elemento no anel, no campo denominado seguinte. A cada processo está associada uma possível fila de mensagens, enviadas por outros processos; isto é, as mensagens são armazenadas no destino. Cada processo possui um nome que é uma cadeia de caracteres, atribuido na hora da sua criação. Os processos têm dois estados possíveis: ativo, ou bloqueado por estar esperando alguma mensagem. Os processos ativos serão ligados entre si formando a fila dos prontos. Para representar estes conceitos foi criado um tipo AnelProc, cuja declaração

```
TYPE AnelProc = POINTER TO DescritorProcesso;
DescritorProcesso = RECORD
        contexto:PROCESS;
        seguinte:AnelProc;          (*anel de processos*)
        basepilha:ADDRESS;
        tamanho:CARDINAL;
        proxmensg:pmensg;           (*fila de mensagens*)
        nome:STRING;
        status:EstadoProc;
        fprontos:AnelProc           (*fila de prontos*)
END;
```

FIGURA IV.4 - DESCRIÇÃO DO TIPO AnelProc

pertence ao módulo de implementação do NucleoTrocaMensg e tem a estrutura ilustrada pela Figura (IV.4):

Vamos apresentar, a seguir, a programação do módulo de definição de NucleoTrocaMensg, ilustrado pela Figura (IV.5) para descrever as primitivas por ele implementadas.

Como no caso anterior trabalharemos com ponteiros para o descritor de processo e para o descritor de mensagem, este último precisando ser exportado pelo núcleo, já que é um argumento das

```
DEFINITION MODULE NucleoTrocaMensg;
  FROM SYSTEM IMPORT WORD;
  FROM ManipulaString IMPORT STRING;
  EXPORT QUALIFIED CriaProcesso, Envia, Recebe, FilaVazia,
      FimProcesso, pmensg, mensg;
  TYPE pmensg = POINTER TO mensg;
  TYPE mensg = RECORD
                 info:WORD;
                 proxmensg:pmensg;
                 origem:STRING
                 END;

  PROCEDURE CriaProcesso (CodigoProcesso:PROC;
      tamanhopilha:CARDINAL; nomep:STRING;):BOOLEAN;
    (*cria um processo associado ao código do procedimento
     CodigoProcesso com uma área de trabalho de tamanho
     tamanhopilha e associa o nome passado como parâmetro ao
     processo criado*)

  PROCEDURE Envia (destino:STRING; pm:pmensg):BOOLEAN;
    (*envia o apontador de descritor de mensagem ao destino
     indicado; devolve valor verdadeiro ou falso dependendo de
     ter conseguido achar o destino*)

  PROCEDURE Recebe (VAR pm:pmensg);
    (*recebe uma mensagem apontada por pm enviada pelo processo
     origem*)

  PROCEDURE FilaVazia : BOOLEAN;
    (*informa se a fila de mensagens está vazia*)

  PROCEDURE FimProcesso;
    (*acaba a execução do processo, retira-o do anel e desloca a
     área de trabalho dele*)
END NucleoTrocaMensg.
```

FIGURA IV.5 - MÓDULO DE DEFINIÇÃO DO NucleoTrocaMensg

primitivas de comunicação. Além da informação própria da mensagem, que é do tipo WORD, no descritor de mensagem está o apontador para a próxima mensagem da fila e o nome de quem enviou a mensagem.

Como a linguagem Modula-2 não possui o tipo STRING nem operações sobre ele, foi necessário supor para a nossa implementação, a existência de um módulo que implemente o tipo STRING, como um array de caracteres, e a operação ComparaString(str1,str2) de tipo BOOLEAN. Na implementação do núcleo aparece então a importação do tipo e da operação a partir

do módulo denominado ManipulaString.

A inicialização do módulo consiste na alocação e inicialização de um descritor de processo, correspondendo ao processo inicial.

Este módulo é utilizado para mostrar novamente a programação do módulo ProdutorConsumidor, no qual são criados os dois processos que se comunicam enviando e recebendo mensagens, usando as primitivas importadas do módulo NucleoTrocaMensg, e cuja listagem ilustrada pela Figura (IV.6) é a seguinte:

```
MODULE ProdutorConsumidor;
  FROM InOut IMPORT WriteInt, Read, WriteText;
  FROM  NucleoTrocaMensg  IMPORT  CriaProcesso,   Envia,  Recebe,
        FilaVazia, pmensg;
  FROM ManipulaString IMPORT STRING;
    CONST N=300;        (*tamanho areapilha*)
    VAR Prod, Cons:STRING;

  PROCEDURE Produtor;
    VAR X:INTEGER; m1:pmensg;
      BEGIN
        LOOP
          Read (X);
          m1 ^.info:=X;
          IF NOT Envia (Cons, m1) THEN HALT
            END;        (*IF*)
      END      (*LOOP*)
   END Produtor;

   PROCEDURE Consumidor;
     VAR X:INTEGER;
         I:CARDINAL
        m2:pmensg;
         Processo:STRING;
     BEGIN
       I:=0;
       LOOP
         I:=I+1;
         Recebe (m2);
         X:=m2 ^.info;
         WriteInt(I,4); WriteInt(X,4); WriteText(m2^.origem);
      END;      (*LOOP*)
   END Consumidor;

BEGIN Prod:='PROD'; cons:='CONS';
  IF NOT CriaProcesso (Produtor, N, Prod) OR
     NOT CriaProcesso (Consumidor, N, Cons) THEN HALT END;
                (*IF*)
     FimProcesso
END ProdutorConsumidor.
```

FIGURA IV.6 - PROGRAMA DO ProdutorConsumidor (2a. VERSÃO)

IV.2.3 CARACTERÍSTICAS PARA O TRATAMENTO DE ENTRADA/SAÍDA E DE INTERRUPÇÕES

IV.2.3.1 MECANISMOS OFERECIDOS PELA LINGUAGEM

Modula-2 foi projetada considerando como um dos seus objetivos, seu uso para a construção de sistemas operacionais.

Foram portanto incluídas facilidades para o tratamento de interrupções e o controle de dispositivos.

Quando um processo chama uma operação de um dispositivo periférico, o processador pode ser alocado a outro processo depois que a operação do dispositivo for iniciada, permitindo desta forma uma execução concorrente entre o processador e o dispositivo. Normalmente, a terminação da operação do dispositivo é assinalada através da interrupção do processador.

Para Modula-2 uma interrupção é considerada uma operação de transferência entre co-rotinas. A transferência é implementada usualmente, de forma preprogramada fazendo parte do pseudo-módulo SYSTEM. Esta combinação é expressa pela chamada a um procedimento:

```
PROCEDURE IOTRANSFER (VAR p1,p2:PROCESS; va:CARDINAL);
```

Analogamente a TRANSFER, esta chamada suspende a co-rotina associada ao dispositivo, atribui seu estado a p1 e transfere o controle para a co-rotina apontada por p2 que estava suspensa. Ela causa também, quando chegar a interrupção da terminação da operação do dispositivo, a suspensão da co-rotina em execução naquele momento (que pode não ser necessariamente aquela que foi ativada pela execução do IOTRANSFER) salva seu contexto apontando-o por p2, e reativa a co-rotina apontada por p1. A variável va é o elemento do vetor de interrupção associado ao dispositivo. O procedimento IOTRANSFER deverá ser importado do pseudo módulo SYSTEM e dependerá da implementação.

No tratamento das interrupções, é necessário que elas sejam inibidas em determinados momentos, como por exemplo para implementar exclusão mútua no acesso a variáveis compartilhadas, ou no caso de serem executadas operações críticas com maior prioridade. Para isto, pode ser associado a cada módulo um determinado nível de prioridade, e a sua execução só pode ser suspensa por uma interrupção de maior prioridade. Quando a prioridade do dispositivo é definida pelo hardware, o nível de prioridade de cada módulo é especificado no seu cabeçalho. Transferências do tipo IOTRANSFER só devem ser usadas dentro de módulos com prioridade especificada e relacionada à prioridade relativa do periférico controlado.

Em [124] são apresentados dois exemplos de gerenciadores

("drivers") de dispositivos, uma impressora e um teclado, programados em Modula-2 usando as características descritas. Para ilustrar mais ainda o uso destas ferramentas, será apresentado na próxima seção um módulo que foi incorporado ao Módulo NucleoHolt para implementar o compartilhamento do processador entre os processos.

IV.2.3.2 MÓDULO FatiadeTempo

Este módulo, local ao módulo NucleoHolt, implementa o conceito de fatia de tempo para uso do processador de forma compartilhada entre os diferentes processos a serem executados. Em lugar de deixar que cada processo monopolize o uso do processador até cedê-lo voluntariamente, por exemplo por causa da execução de uma primitiva Espera, é atribuido a cada processo uma fatia de tempo fixa para o uso do processador.

Para implementar este exemplo foi considerado o relógio KW11-L do PDP-11 que interrompe a cada 16,67mseg.

Este módulo precisa importar tipos e procedimentos do pseudo módulo SYSTEM, do módulo NucleoHolt, e do GerenteMemoria. A variável pc é do tipo Filaproc e é importada do módulo de implementação de NucleoHolt, no qual é declarado o módulo FatiadeTempo; ela representa a cabeça da fila dos prontos. O módulo implementa uma co-rotina chamada Relogio, que a cada 16,67 msg interrompe o processo que esteja executando, e passa o controle do processador ao próximo processo na fila dos prontos, colocando o processo interrompido no final da fila.

Vamos apresentar agora a programação do módulo FatiadeTempo ilustrada pela Figura (IV.7).

Na inicialização do módulo, depois de ser alocada a área para o processo Relógio, este processo é criado e é transferido (através de TRANSFER) o controle para ele. A primeira instrução que ele executa é novamente uma transferência (IOTRANSFER) para o processo original que foi salvo em Procint. Neste primeiro caso o controle volta para a inicialização do módulo, e são executadas duas instruções para inicializar o relógio e possibilitar suas interrupções. Quando acontecer a interrupção do relógio, será reativada a co-rotina Relógio e será executado o código para

```
MODULE FatiadeTempo;
  IMPORT PROCESS,  NEWPROCESS,  ADDRESS, WORD, IOTRANSFER, Aloca,
         FilaProc, DescritorProcesso, pc;
    VAR  Rel, Procint : PROCESS;
         S[7775468]:BITSET;        (*endereço do registrador do
                                     relógio KW11-L do PDP-11*)
         PilhaRelogio : ADDRESS;
    CONST TamanhoPilha := 128;

  PROCEDURE Relogio;     (*atua como co-rotina*)
    VAR eu, p1 : FilaProc;
    BEGIN
      LOOP IOTRANSFER (Rel, Procint, 100B);
           (*100B  é o endereço associado ao vetor de  interrupção
            do relógio*)
           (*espera acontecer interrupção*)
      IF pc^.seguinte # NIL THEN
         eu:=pc;
         pc:=pc^.seguinte;
         p1:=pc;eu^.seguinte:=NIL;eu^.contexto:=Procint;
         WHILE p1^.seguinte # NIL DO
           p1:=p1^.seguinte
         END
         p1^.seguinte:=eu;
         procint:=pc^.nome
      END;   (*IF*)
    (*tirou o processo interrompido,  o colocou no final da  fila
     dos prontos, e ativou o seguinte*)
      EXCL(S,7)       (*reinicializa o relógio*)
    END;     (*LOOP*)
  END Relógio;
  BEGIN
   Aloca (PilhaRelogio,TamanhoPilha);
   NEWPROCESS (Relogio, PilhaRelogio, TamanhoPilha, Rel);
   TRANSFER (Procint, Rel);
   EXCL(S,7);    (*cancela pedido de interrupção*)
   INCL(S,6)     (*possibilita interrupções*)
END FatiadeTempo.
```

FIGURA IV.7 - PROGRAMA DO MÓDULO FatiadeTempo

fazer a troca dos processos na fila de prontos, como já foi explicado. No final o relógio é reinicializado, e como o procedimento é cíclico, é executada novamente a chamada IOTRANSFER, passando o controle do processador para o processo escolhido e suspendendo a co-rotina novamente à espera de interrupção.

IV.2.4 NÚCLEO DE MULTIPROGRAMAÇÃO PARA O PROTOCOLO DE CHAMADA REMOTA DE PROCEDIMENTO

A partir dos núcleos já mencionados, o de WIRTH [70] e o NucleoHolt [143], que implementam o conceito de processos concorrentes compartilhando variáveis globais e cujo acesso só pode ser feito com exclusão mútua, foi desenvolvido por DA SILVA

Este núcleo foi projetado visando a implementação do ambiente de programação distribuída. Num sistema distribuído cada nó físico ou estação suportará a execução multiprogramada de vários processos através de um núcleo próprio. Como foi mencionado na introdução, o nosso modelo para um ambiente de programação distribuída usando a linguagem Modula-2 se baseia no uso de RPC para a comunicação e sincronização entre processos localizados em máquinas distintas. Portanto o núcleo foi projetado, especialmente voltado para atender às necessidades do Protocolo de Chamada Remota de Procedimento (PCRP), que será apresentado mais adiante neste capítulo.

O módulo chamado NUCLEO cuja listagem pode ser analisada em [45], e cujo módulo de definição apresentaremos a seguir na Figura (IV.8), provê facilidades para multiprogramação à base de filas de eventos e partição de tempo, sem prioridade para o escalonamento. Ele apresenta algumas características que gostaríamos de salientar resumidamente, já que a listagem do módulo de definição pode ser considerada autoexplicativa.

- O módulo exporta primitivas classificadas em duas categorias principais: criação e destruição de processos, e sinalização de processos.

As **primitivas de criação e destruição de processos**, em particular Exec e Exit, permitem que o usuário implemente a sua própria política de otimização de memória através do manuseio do endereço da área de trabalho, associado ao processo.

As **primitivas de sinalização** Wait e Signal são análogas às dos outros núcleos. Foi implementada também a primitiva Timeout que tem como objetivo delimitar o tempo de espera por um sinal.

O NUCLEO oferece outras **primitivas de apoio**, como WaitIO, que permite ao processo se suspender esperando por uma interrupção de E/S, e facilita a implementação de controladores de dispositivos. A primitiva Delay atrasa o corrente processo por um tempo determinado pelo parâmetro de entrada. Foram implementadas também **primitivas para manipulação de erros**, tais como SetError, Error e GetError.

```
DEFINITION MODULE NUCLEO
FROM SYSTEM IMPORT BYTE, ADDRESS;
EXPORT QUALIFIED SIGNAL,  DWORD,  InitSignal,  Exec,  Exit, Kill,
                 Signal,  Wait,  WaitIO,  Timeout,  Empty, Delay,
                 SetError,  GetError,  Error,  ProcID,  TimeID,
                 ContID, CompareDW, MaxNumProc;

CONST MaxNumProc = 256;
(*Máximo número de processos gerenciados pelo sistema*)
TYPE SIGNAL;
(*SIGNAL é o meio de sincronização entre os processos*)
TYPE DWORD;
(*DWORD é uma palavra dupla de 32 bits*)

PROCEDURE Exec (pro : PROC;
                adr : ADDRESS;
                wsp : CARDINAL);
(*Reserva  uma área de memória de wsp bytes para o novo processo,
cujo código está no procedimento pro, e transfere o controle para
ele.  Caso  adr igual a NIL,  o núcleo aloca  automaticamente  um
endereço no heap*)

PROCEDURE Exit  (VAR s   : SIGNAL;
                 VAR adr : ADDRESS;
                 VAR wsp : CARDINAL);
(*Cancela o processo corrente,  envia sinal e retorna o espaço de
memória liberado*)

PROCEDURE Kill;
(*aborta o processo corrente*)

PROCEDURE Signal (VAR s: SIGNAL);
(*Se  nenhum processo está esperando por s, Signal não tem nenhum
efeito.  Caso  contrário,  um processo que esteja esperando por s
ganha o controle e continua a execução depois do Wait*)

PROCEDURE Wait (VAR s: SIGNAL);
(*O  corrente processo entra em estado de espera até  que
outro processo envie o sinal s*)

PROCEDURE Timeout (VAR s:SIGNAL; t:CARDINAL):BOOLEAN;
(*O  processo corrente entra em estado de espera  até  que  algum
outro  processo envie o sinal s ou que se esgote o tempo t.  Se o
sinal  s chegar antes de esgotar o tempo t,  o processo  ganha  o
controle e Timeout retorna FALSE.  Se o sinal não chegar,  ao fim
do tempo t o processo ganha o controle e Timeout retorna TRUE*)

PROCEDURE WaitIO (intr: CARDINAL);
(*O processo corrente entra em estado de espera até a interrupção
intr ocorrer*)

PROCEDURE Empty (s: SIGNAL): BOOLEAN;
(*testa  se  algum  processo está esperando  por  um  determinado
sinal*)

PROCEDURE Delay (t: CARDINAL);
(*atrasa o processo corrente por um tempo t*)

PROCEDURE InitSignal (VAR s: SIGNAL);
(*Um  objeto do tipo SIGNAL deve ser inicializado por  InitSignal
antes  de  ser  empregado  em  qualquer  outra  operação.  Após
InitSignal(s) Empty(s) é FALSE*)

PROCEDURE SetError (e:CARDINAL);
(*Permite  que procedimentos compartilhados por vários  processos
possam passar erro para o processo chamador.  Por omissão o  erro
tem valor zero*)

PROCEDURE Error ():BOOLEAN;
(*Retorna  TRUE  se e somente se o código de erro é diferente  de
zero*)

PROCEDURE GetError (VAR e: CARDINAL);
(*GetError sempre re-inicializa o erro com zero.  Para propagar a
exceção deve-se chamar SetError após GetError*)

PROCEDURE ProcID ():BYTE;
(*devolve o identificador do processo*)
```

```
PROCEDURE TimeID ():DWORD;
(*TimeID gera um identificador de tempo de 32 bits, em função do
ano, mês, dia, hora, minuto e décimo de segundo. Duas ou mais
chamadas, dentro de um intervalo menor que 1 décimo de segundo,
retornam o mesmo identificador. Do contrário, um certo TimeID
terá sempre valor maior que o anterior, garantidamente até o fim
do ano de 1990*)

PROCEDURE ContID ():DWORD;
(*Contador gerado pelo sistema para cada processo, que consiste
de um número de 32 bits que é sempre inicializado a partir de um
TimeID, automaticamente gerado na criação do processo. A cada
chamada a ContID o contador é decrementado*)


PROCEDURE CompareDW (VAR w1,w2:DWORD):INTEGER;
(*Próprio para ser empregado na comparação de identificadores
gerados por ContID e TimeID (vide acima)*)

END NUCLEO.
```

FIGURA IV.8 - MÓDULO DE DEFINIÇÃO DO NÚCLEO DE MULTIPROGRAMAÇÃO UTILIZADO NA IMPLEMENTAÇÃO DE RPC

Por necessidades que serão apresentadas na próxima seção, foram implementadas **primitivas para geração e comparação de identificadores.** Todas as primitivas do NUCLEO são visíveis para os outros módulos poderem importá-las, e o NUCLEO é um módulo de biblioteca livre e de emprego geral; isto é, qualquer módulo envolvido ou não com chamada remota pode fazer uso de seus serviços.

Na implementação do módulo NUCLEO podemos identificar a existência do módulo local Dispatcher, análogo ao módulo FatiadeTempo descrito anteriormente. Uma diferença importante a ser levantada, é a necessidade da existência de um processo ocioso que tomará o controle no caso de todos os demais processos estarem bloqueados. No caso anterior, executando num único processador, essa situação indicava bloqueio perpétuo. Estendendo o núcleo para o caso distribuído, supondo a existência de vários processadores ligados, esta situação pode acontecer, e neste caso o processo ocioso executará até algum processo sair do seu estado bloqueado.

Esta versão do núcleo foi implementada para executar no microcomputador PC/XT ou compatível, em ambiente MS/PC DOS 2.X ou 3.X. Parte do seu código, em particular o tratamento de interrupções, é dependente dessa arquitetura.

IV.3 CHAMADA REMOTA DE PROCEDIMENTO

Como já foi mencionado no Capítulo II existe uma variedade grande de sistemas distribuídos com características distintas. Uma forma de simplificar a abordagem dos problemas a serem resolvidos na construção destes sistemas, é observando que na sua maioria eles têm uma estrutura do tipo cliente/servidor para a interação entre os seus componentes.

Embora que estejam surgindo alguns modelos padrão de RPC [36, 53], existe uma certa variedade de seus projetos e suas implementações. Os diferentes mecanismos de RPC não diferem somente na representação dos parâmetros ou nos protocolos de transporte subjacentes. Eles diferem também na maneira como os componentes são ligados, na riqueza e flexibilidade das ligações, e na maneira na qual as estruturas de sistemas e as semânticas das chamadas forçam a divisão de estados entre clientes e servidores. Estas considerações mostram como é difícil prover uma forma automática de interligar mecanismos de RPC, não sendo entre aqueles que apresentam fortes similaridades. É necessário então que coincidam, em clientes e servidores, os tipos de dados e representações de parâmetros e resultados, que possam ser passadas as mesmas estruturas de dados e que as semânticas das chamadas e os mecanismos de ligação sejam iguais.

Sendo assim, a escolha do modelo de RPC afetará significativamente o projeto das aplicações distribuídas. É nosso objetivo aqui analisar as características principais deste mecanismo, levantar alguns problemas que não foram resolvidos ainda e apresentar o nosso modelo de RPC a ser acrescentado à linguagem Modula-2.

IV.3.1 CARACTERÍSTICAS

IV.3.1.1 O MECANISMO

A chamada remota de procedimento provê um mecanismo de comunicação entre componentes de programa que trocam dados e se sincronizam. O mecanismo se assemelha fortemente com a chamada de procedimento das linguagens convencionais salvo algumas ressalvas: a mais forte é que quem chama, e quem é chamado,

(normalmente nomeados de cliente e servidor), estão em espaços de endereçamento diferentes, geralmente ligados a máquinas distintas. Por causa desta semelhança, alguns modelos integram o mecanismo de RPC na linguagem de programação, de maneira que as chamadas locais e remotas sejam sintaticamente iguais; esta propriedade é chamada de **transparência sintática**. Embora esta transparência seja frequentemente conseguida, a **transparência semântica** é raramente obtida nas linguagens de programação atuais.

As chamadas locais diferem em vários aspectos das chamadas remotas. As chamadas remotas são mais vulneráveis a falhas, já que elas envolvem pelo menos outro componente de programação e provavelmente outra máquina e uma rede. Isto faz com que os programas que fazem uso de RPC precisam tomar conta de falhas que não acontecem nas chamadas locais. Por exemplo, os clientes de RPC podem sobreviver a falhas dos servidores; chamadas remotas podem continuar a sua execução depois de falhas dos clientes, podem adquirir o estado de máquinas remotas que sobrevivam a falhas dos clientes; podem acontecer falhas na rede, na transmissão da chamada ou na recepção dos resultados, depois do servidor ter executado a operação chamada.

Outra diferença reside no fato de que geralmente as referências externas a procedimentos locais são resolvidas por um ligador antes de iniciar a execução, enquanto que as referências externas a procedimentos remotos devem ser resolvidas em tempo de execução. Nos casos mais simples o suporte de RPC pode tomar ações por omissão ("default actions"), mas nos casos onde é necessário um controle mais preciso sobre onde se encontra o procedimento remoto, o cliente deverá utilizar rotinas da biblioteca de suporte em tempo de execução, para realizar essa localização e a ligação.

Nas chamadas remotas, é necessário ter informação adicional para a construção de tipos de dados que possam ser transmitidos entre as máquinas, por exemplo, no caso das máquinas terem representações diferentes para os mesmos tipos de dados e precisarem traduzir estas para um tipo intermediário comum às duas máquinas. Finalmente, as chamadas remotas ocasionam uma sobrecarga significativa em relação às chamadas locais, que pode ter uma influência não desprezível no projeto dos programas que

as usam.

Por estas razões, o maior problema em integrar o mecanismo de RPC nas linguagens reside na necessidade de conciliar a semântica das chamadas remotas com das chamadas locais. O mecanismo de RPC é geralmente implementado da seguinte forma, como é ilustrado pela Figura (IV.9):

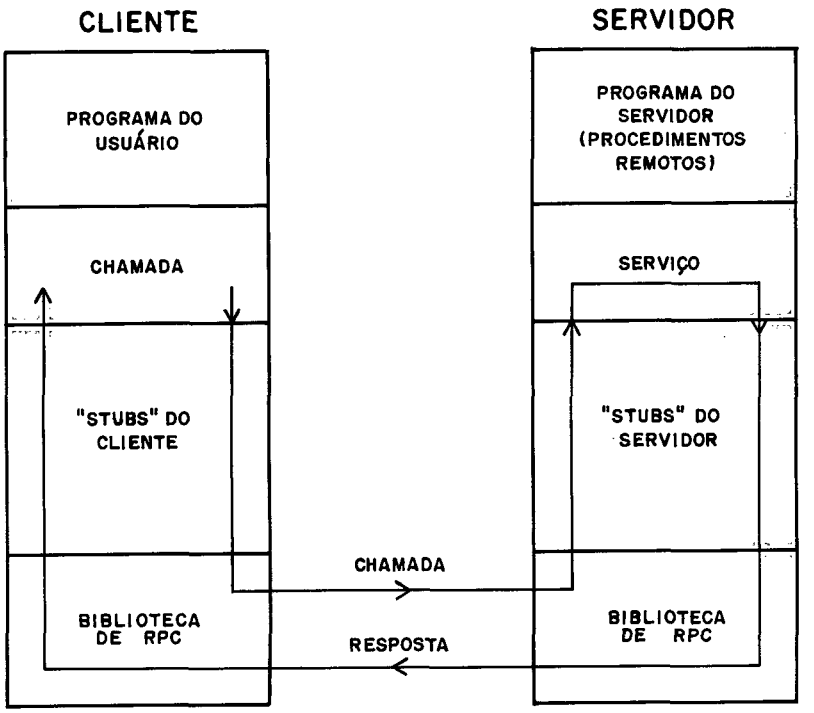

FIGURA IV.9 - MECANISMO DE CHAMADA REMOTA DE PROCEDIMENTO

As duas partes da aplicação estão divididas através de uma fronteira procedimental, e um procedimento "testa de ferro", com o mesmo nome daquele que está no servidor, é colocado no processo cliente. O procedimento "testa de ferro" do cliente, geralmente chamado de "stub", é responsável por receber os parâmetros da chamada e empacotá-los num formato de transmissão adequado antes de enviá-los ao servidor. O "stub" do cliente espera a resposta do servidor e desempacota os resultados antes de passá-los na representação local de volta para o procedimento que fez a chamada. Do lado do servidor a situação é similar, tendo também um "stub" correspondente a cada interface exportada, com a

exceção de que o servidor é capaz de servir vários tipos de chamadas. Do lado do servidor existe um programa contendo um laço que espera uma mensagem e decide que procedimento chamar; o "stub" correspondente desempacota os parâmetros para o formato local e dispara um processo que chama o procedimento servidor. Quando o procedimento acabar a sua execução, seus resultados são empacotados e transmitidos de volta ao cliente pelo "stub" correspondente. É interessante salientar que existe uma única linha de controle no programa de aplicação, apesar de existirem duas máquinas e dois processos no sistema. Deve-se notar também que, como está se supondo que o cliente e o servidor têm espaços de endereçamento diferentes, todos os parâmetros e resultados devem ser passados por valor.

Um dos propósitos mais importantes dos "stubs" é traduzir a representação dos tipos de dados da máquina local no da máquina remota e vice versa. Isto é normalmente realizado utilizando uma respresentação intermediária, independente de máquina com o objetivo de transmitir e traduzir para as convenções locais de cada máquina.

Os "stubs" são gerados de duas maneiras: pelo programador ou automaticamente. No caso normal o implementador provê um conjunto de funções de tradução a partir das quais o usuário pode construir seus próprios "stubs". A responsabilidade da consistência dos parâmetros do procedimento entre cliente e servidor é inteiramente do usuário.

A geração automática de "stubs" é frequentemente realizada a partir de uma linguagem de descrição de parâmetros, que define as interfaces entre cliente e servidor em termos de procedimentos e seus parâmetros, construidos a partir dos tipos básicos. Esta definição é processada para gerar automaticamente os "stubs" apropriados, que depois podem ser compilados e ligados na forma normal com o código do cliente e do servidor.

Uma forma de geração automática de "stubs" será analisada com mais detalhe na apresentação de nossa proposta numa próxima seção deste capítulo.

Achamos importante levantar aqui algumas características comuns entre o conceito de RPC e o de portas, apresentado no capítulo anterior. Podemos fazer uma analogia entre as RPC e os envios de mensagens para portas de saída locais ou portas de

entrada remotas, particularmente para os envios bloqueados de mensagens. Os dois modelos possuem como objetivo principal oferecer modularidade para a construção dos sistemas. Outra característica análoga é a de ligação de portas entre módulos e a ligação dos procedimentos remotos dos servidores com os módulos clientes, que será analisada com mais detalhes nas próximas seções.

NELSON em [116] define resumidamente cinco características que ele considera essenciais para o mecanismo de RPC:

i) a semântica das chamadas define o comportamento abstrato da invocação do mecanismo;

ii) a ligação e configuração estabelecem a nomeação e a interconexão dos programas que se comunicam através de RPC;

iii) o controle de tipos reforça a compatibilidade das ligações entre programas;

iv) a funcionalidade dos parâmetros determina as restrições, quando existem, nos parâmetros passados no mecanismo de RPC;

v) as interações entre o mecanismo de RPC e qualquer mecanismo independente de processamento paralelo e de tratamento de exceções.

Estas características são essenciais para o mecanismo de RPC, cujo objetivo é transparência local e remota no nível da linguagem em um sistema distribuido programado homogeneamente. Mas se a exigência da transparência no nível da linguagem é relaxada, ele sugere uma divisão destas características em dois grupos.

O primeiro grupo contém as características fundamentais de invocação. A semântica da chamada e o controle de concorrência são essenciais para qualquer mecanismo de invocação remota, seja este baseado em procedimentos, mensagens ou alguma outra forma de comunicação. Estas duas características são fundamentais porque determinam a semântica do comportamento de baixo nível das primitivas de invocação.

O segundo grupo contém as características que determinam a transparência entre chamadas locais e remotas, no nível da linguagem. Estas são ligação e configuração, teste de tipo e

funcionalidade dos parâmetros.

Em [116] NELSON aponta que as características mencionadas não garantem que a inclusão de RPC numa linguagem de programação poderosa façam com que o uso de RPC seja apropriadamente utilizado. É necessario considerar também, outras características mais ligadas ao ambiente de programação do que à linguagem, que ele denomina de secundárias em relação às anteriores. Podemos citar a importância da RPC ter um bom desempenho, a necessidade das interfaces estarem bem projetadas, o uso de transações atômicas para programação robusta na presença de colapsos, e a consideração da autonomia dos nós individuais quando o desejo de transparência entra em conflito com a demanda de descentralização. Por último, precisa-se de ferramentas para depuração remota, já que é essencial que os programadores não precisem procurar em máquinas fisicamente distintas, para depurar os programas distribuidos.

Passaremos então a analisar com mais detalhe particularmente as características apontadas como essenciais, mas dentro do possível, procuraremos separar os problemas de configuração que serão abordados nos capítulos seguintes.

IV.3.1.2 PASSAGEM DE PARÂMETROS E INTERFACES

O fato do cliente e servidor estarem localizados em espaços de endereçamento diferentes, e frequentemente em máquinas distintas, inviabiliza a passagem de apontadores e de parâmetros por referência. A maioria dos mecanismos de RPC passa os parâmetros por valor, isto é, os parâmetros e os resultados são copiados entre o cliente e o servidor através da rede de comunicação. A transmissão de tipos compactos tais como inteiros, contadores e arranjos pequenos, geralmente não ocasionam problemas. Mas é bem mais custoso passar arranjos grandes ou multidimensionais, já que seria consumido um tempo enorme na transmissão de dados, quando nem sempre seriam todos utilizados. Em alguns mecanismos de RPC existem limitações no tamanho dos parâmetros e resultados passados pela rede de transmissão de mensagens subjacente. Podemos tirar então como conclusão da semântica da passagem de parâmetros, que é às vezes necessária uma reestruturação cuidadosa das interfaces a fim de que os

parâmetros se tornem mais específicos e envolvam uma transmissão de dados mínima.

Precisa-se tomar cuidado também com a passagem como parâmetros de tipos de dados que contêm apontadores embutidos, por exemplo podemos citar uma lista de elementos ligados. Em [173] WILBUR e BACARISSE sugerem duas soluções alternativas para este caso. A primeira consiste em transformar a lista em um arranjo contendo os elementos da lista. A segunda é identificar as operações realizadas sobre a lista, de maneira que seja possível construir uma interface que represente um tipo de objeto mais do que um tipo de dado.

Poucos mecanismos de RPC permitem a passagem de parâmetros por referência, que só pode ser realizada em sistemas nos quais todos os processos compartilham o mesmo espaço de endereçamento. O suporte para o controle das páginas de memória referenciadas é complicado e provoca uma sobrecarga potencialmente não desprezível. Deve-se notar que neste caso é necessário que o sistema distribuído seja homogêneo, tanto o hardware quanto o software, isto é, que as máquinas e os sistemas operacionais sejam idênticos.

Outro problema analisado em [173] é ocasionado pela passagem de procedimentos como parâmetros. Geralmente o sistema distribuído não suportará acesso a variáveis globais e portanto não faz sentido mandar o código de um procedimento a ser executado num outro contexto, e muito menos ainda no caso de ambientes heterogêneos. Vamos supor que a implementação permita que um cliente possa atuar como servidor também, especialmente quando o cliente está bloqueado esperando o resultado da chamada. Neste caso o cliente pode passar um "handle" a um servidor, dentro do mesmo cliente, que implemente o procedimento requerido no lugar de passar o código mesmo. Quando o servidor precisar chamar o procedimento, ele se transforma em cliente, faz uma chamada a seu cliente original que, temporariamente, se comportará como servidor. A semântica desta operação não é exatamente igual a de passar um procedimento como parâmetro ao servidor, mas é mais simples de implementar e poderosa para usar. A possibilidade do servidor poder chamar seu cliente é muito importante, e precisa-se tomar cuidado no projeto do protocolo de RPC para garantir esta possibilidade.

IV.3.1.3 SEMANTICA DAS CHAMADAS

Se por um lado os sistemas distribuidos oferecem um maior paralelismo, por outro eles são mais propensos à ocorrência de falhas parciais. Em tais sistemas é possível que o cliente ou o servidor falhe independentemente, e que seja depois reinicializado. A semântica da chamada determina quantas vezes o procedimento remoto pode ser executado no caso de ocorrerem falhas. Apresentaremos aqui a caracterização das diferentes semânticas segundo os critérios escolhidos em [173], que são os mais utilizados pelos pesquisadores desta área, apesar de reconhecer a falta de consenso em alguns pontos.

A semântica mais fraca é denominada de **possível** ("possibly" ou "maybe"), como foi definida por SPECTOR em [154], e consiste numa única tentativa de chamada. Na ocorrência de erros, estes são transmitidos ao cliente. Para a sua implementação, a forma mais adequada é através da especificação de temporização ("timeout") para cada chamada, de maneira que o mecanismo de RPC a abandone se não for possível que seja completada durante o intervalo de tempo requerido, como é sugerido por LISKOV et alii em [100].

Isto corresponde ao caso do cliente enviar uma mensagem ao servidor sem esperar por uma resposta ou uma mensagem de reconhecimento. Esta semântica é utilizada em algumas aplicações que utilizam redes locais com probabilidade muito alta de transmissão bem sucedida. Portanto, a maioria dos erros que podem acontecer são específicos da aplicação, causados pela não disponibilidade ou sobrecarga do servidor. Mas esta semântica não é apropriada para o caso de mecanismo de RPC transparente.

A semântica denominada **pelo-menos-uma-vez** ("at-least-once") é implementada da seguinte forma: se a resposta não chegar dentro de um intervalo de tempo pré-determinado, o cliente retransmite o pedido. Na Figura (IV.10) são ilustrados os diferentes casos que podem acontecer, e fica clara a impossibilidade de determinar, no caso de falha, se ela foi devida ao colapso do servidor, ou à perda da chamada ou da resposta.

Podemos notar que o procedimento no servidor pode ser executado várias vezes ou até só parcialmente. Este comportamento é diferente do da chamada de procedimento local mas pode ser

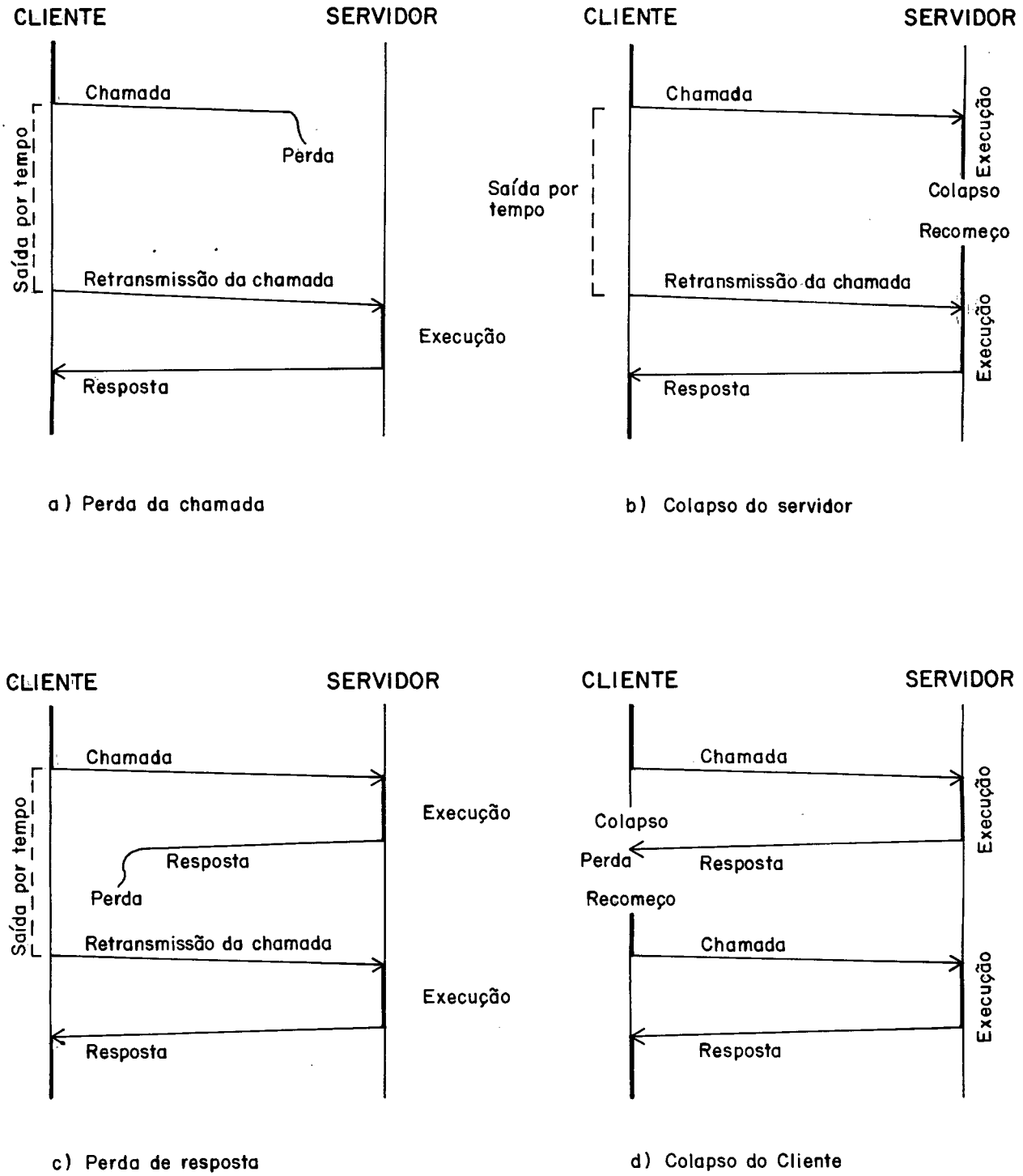

FIGURA IV.10 - FALHAS POSSÍVEIS NA COMUNICAÇÃO ENTRE PROCESSOS

implementado por um protocolo simples. Se for utilizado tal método, e então perdida a transparência semântica e o servidor precisa ser projetado utilizando funções **idempotentes**. Isto implica que o programador deve garantir que execuções múltiplas da mesma chamada, gerem o mesmo resultado que o de uma chamada única com os mesmos parâmetros.

Dentro desta caracterização, NELSON [116] diferencia uma outra semântica denominada **última-de-várias** ("last-of-many"), na qual é garantido ao cliente, que mesmo tendo repetido várias vezes a chamada, ele recebe os resultados da última execução. Para isto é necessário casar os números de sequência gerados para as mensagens de chamada, com os das mensagens de resultados. Na ocorrência de falhas esta semântica coincide com a da chamada de procedimento local.

Uma semântica mais adequada é a denominada de **no-máximo-uma-vez** ("at-most-once"). Neste caso é garantido que as funções do servidor serão executadas uma vez, na ausência de colapso do servidor. O protocolo subjacente precisa descartar pedidos de chamadas duplicados e retornar resultados prévios quando as mensagens de resposta forem perdidas. O protocolo é mais complexo e precisa guardar resultados de chamadas anteriores. Se o servidor entrar em colapso, o procedimento pode ter sido executado parcialmente, e o programa de aplicação ou o operador precisa reestabelecer a consistência dos dados que foram modificados.

A semântica mais poderosa é a denominada **exatamente-uma-vez** ("exactly-once"), quando é garantido que o procedimento é executado uma única vez, mesmo se o servidor entrar em colapso. Para conseguir este comportamento é geralmente necessário prover ações atômicas, mas esta área está ainda em estudo. Na ausência de falhas, esta semântica coincide com a da chamada local de procedimento. **Para obter uma semântica idêntica para chamadas locais e remotas, ou seja, para construir um mecanismo de RPC transparente, é necessário implementar a semântica exatamente-uma-vez na ausência de falhas e a semântica última-de-várias na presença de falhas**, como foi feito no sistema Emissary de NELSON [116].

Apesar da semântica no-máximo-uma-vez ser mais desejável e

relativamente simples de implementar, alguns mecanismos de RPC oferecem unicamente a semântica pelo-menos-uma-vez. Esta é muito fácil de implementar, necessitando essencialmente de um contador de tempo ("timer") e de um mecanismo para repetir a chamada, embutido nas rotinas de transmissão do cliente. Num sistema operacional multiprogramado, estas facilidades podem ser providas no espaço de endereços do usuário em vez de estar no núcleo do sistema operacional, de maneira a se obter um pacote de RPC mais trnasportável. A maior desvantagem desta semântica é a de forçar os programadores da aplicação a projetar interfaces idempotentes, como já foi mencionado anteriormente.

Enquanto a semântica pelo-menos-uma-vez precisa de um protocolo mais simples de implementar e minimiza a quantidade de informação do estado do cliente que deve ser armazenado no servidor, em alguns casos ela força a definição de interfaces mais complexas.

A semântica no-máximo-uma-vez é a mais utilizada pelas implementações de RPC. Esta se aproxima muito do comportamento de chamada de procedimento local, exceto quando acontecem erros não recuperáveis que precisam ser comunicados ao cliente ou ao servidor, tais como colapso de um deles durante a chamada. O problema de tratamento de erros será analisado a seguir.

IV.3.1.4 TRATAMENTO DE ERROS

O mecanismo necessário para dar suporte a uma RPC é muito mais complexo que para chamada local, já que ele envolve vários componentes de software e hardware, sendo que cada um pode falhar de forma independente. Somente nos casos mais simples é aceitável tratar estas falhas da mesma maneira que são tratadas falhas de hardware, abortando o programa que fez a chamada. O programador deve dispor de alguma forma de detectar e identificar os diferentes erros e tomar alguma ação corretiva.

Dependendo de se o mecanismo de RPC é transparente ou não, existem formas diferentes de tratamento de erros. Quanto maior a transparência do mecanismo, maior é a responsabilidade da implementação de RPC para cuidar das falhas possíveis do sistema. Quando ocorre um erro na comunicação usando RPC entre um cliente

e um servidor, a implementação pode tentar a recuperação do erro, ou pode passar a indicação de erro para o nível da aplicação levantando uma exceção. Por exemplo, nas duas implementações de RPC transparente, no projeto do sistema Emissary [119] e no sistema CEDAR [19, 162], os erros são tratados na própria implementação, sempre que seja possível. No sistema CEDAR, entretanto, quando acontecem certos erros considerados fatais, eles são indicados no nível da aplicação, mas somente depois de ter sido tentada a sua recuperação durante um certo intervalo de tempo. Em alguns casos a escolha da melhor resposta para um erro depende da aplicação. Por exemplo, quando é chamado um servidor que está congestionado, é importante não continuar chamando-o persistentemente, já que isto lhe provocará uma sobrecarga ainda maior. Entretanto voltar ao nível mais alto da aplicação tem um custo, já que acarreta um atraso grande para completar a chamada. Por outro lado, a passagem de erros para o nível da aplicação permite que sejam tomadas ações específicas da aplicação de forma imediata. No exemplo anteriormente citado, se fosse passada alguma indicação de congestionamento ao cliente do servidor, ele poderia contactar outros servidores alternativos.

A maioria das linguagens de programação não provê facilidades para detectar e manipular os erros que podem acontecer numa RPC. Somente linguagens como MESA, ADA e CLU provêem mecanismos para tratar erros definidos pelo usuário, de forma elegante e simples. Em outros casos, onde o mecanismo de RPC é provido como uma extensão da linguagem, são definidos manuseadores para várias classes de erros.

Existem determinados tipos de erros que não se consegue detectar, e são somente percebidos como chamadas remotas que não retornam resultados num tempo razoável. Neste caso somente o cliente poderá escolher qual é o intervalo de tempo adequado para a temporização. Mas esta solução pode ser considerada viável unicamente sob o ponto de vista do desempenho do sistema. NELSON [119] aponta que qualquer que seja o nível de abstração no qual sejam interceptados os erros, é desejável obter informação apropriada sobre eles para poder tomar decisões inteligentes para o seu tratamento.

Geralmente o sistema de suporte de execução de RPC enxerga uma variedade de falhas diferentes. Por exemplo, a máquina alvo

pode entrar em colapso, o servidor pode não entender a mensagem, ou a resposta pode não conseguir voltar ao cliente, mas o que interessa ao programador é saber se o procedimento remoto foi executado ou não. Quando é utilizada a semântica no-máximo-uma-vez, podem existir dúvidas, já que a situação na qual o servidor entra em colapso imediatamente depois de ter recebido o pedido de uma chamada não é distinguível da situação na qual isto acontece imediatamente antes de responder à chamada, depois de tê-la executado.

Outro problema que precisa ser considerado pelas implementações de RPC é a necessidade de abortar execuções de procedimentos, isto é, parar uma execução e destruir a sua computação. Isto será utilizado em situações relacionadas com colapso do cliente, nas quais podem ter sido aceitas chamadas suas antes de seu colapso, e o servidor precisar abortar a execução do procedimento. Neste caso o servidor precisa ser notificado da falha do cliente, e mais ainda quando o cliente retém algum recurso essencial do servidor. Se o cliente entra em colapso ou abandona a chamada, o recurso precisa ser liberado automaticamente. Por causa da assimetria inerente à chamada de procedimento este não é um problema simples. No caso do servidor estar executando um procedimento de um monitor, para este ser abortado precisa-se tomar cuidado de liberar a exclusão mútua e de preservar os invariantes do monitor.

Uma chamada remota originada, diretamente ou indiretamente (através de chamadas aninhadas), por um processo alocado a um nó que entra em colapso é denominada de **órfão**. Para manter a transparência semântica das chamadas é necessário que na presença de falhas, as chamadas remotas sejam implementadas com a semântica última-de-várias. NELSON sugere em [119], que como as chamadas de procedimentos atômicas são muito caras para ser o mecanismo de implementação de RPC, a maneira mais barata de obter esta semântica é através do extermínio automático de órfãos depois de detetadas as falhas. Ele desenvolveu para o sistema Emissary algoritmos complexos para deteção e extermínio de órfãos a fim de garantir a semântica mencionada. Este problema será tratado novamente quando forem analisadas as dificuldades que ainda subsistem na implementação de RPC.

IV.3.1.5 LIGAÇÃO

Num sistema distribuído, a **ligação** é o processo de identificar e ligar todos os módulos de um programa distribuído. Este processo está muito relacionado com a forma de configuração do sistema, mas procuraremos fazer uma abordagem o mais independente possível aqui, já que o tema de configuração será tratado nos próximos capítulos.

Tanto nos sistemas convencionais quanto nos distribuídos existe um amplo espectro de tempos de ligação possíveis. Quanto mais cedo for feita a ligação, mais estática e possivelmente mais eficiente ela será, mas apresenta a desvantagem de não ser flexível. Por outro lado, quanto mais retardado for o processo de ligação, mais dinâmico e flexível ele é, mas pode apresentar um desperdício de espaço de memória e baixo desempenho.

Podemos diferenciar, tal como foi feito em [119], **quatro tempos de ligação** que apresentaremos em ordem crescente de flexibilidade:

i) em **tempo de compilação** as ligações podem ser definidas quando os módulos são compilados.

ii) **antes da carga e antes da execução**, podem ser definidas as ligações quando os módulos são combinados para formar uma determinada configuração.

iii) em **tempo de carga**, mas antes da execução podem ser estabelecidas ligações estáticas.

iv) em **tempo de execução** podem ser estabelecidas ligações dinâmicas que estão sujeitas a mudanças durante essa mesma execução.

No caso específico de RPC é necessário que o cliente conheça a localização do servidor antes que seja executada a chamada remota de procedimento. O processo pelo qual o cliente obtém essa informação é chamado de **ligação**. Esta ligação é superficialmente

análoga à ligação de módulos compilados separadamente das linguagens de programação convencionais, mas pode diferir particularmente em três aspectos como foi apontado por WILBUR e BACARISSE em [173]:

i) a ligação pode ser feita em qualquer momento anterior à invocação da chamada, e em muitos casos é deferida até o tempo de execução.

ii) a ligação pode mudar durante a execução do programa do cliente.

iii) é possível ligar um cliente a vários servidores similares simultaneamente.

A forma como estas diferenças são exploradas por uma implementação afeta enormemente a forma na qual o usuário percebe o mecanismo de RPC.

Analisando com mais detalhe o processo de ligação podemos considerar dois aspectos. Por um lado é necessário especificar com quem o cliente quer se ligar e por outro determinar o endereço do procedimento que está sendo chamado. Estes dois aspectos são chamados de **nomeação e localização.**

O processo de ligação oferecido por uma implementação de RPC precisa ligar o importador de uma interface com um dos seus exportadores. Depois de feita a ligação, o importador pode chamar os procedimentos implementados pelo exportador remoto. Geralmente o nome da interface consiste de duas partes: o tipo e a instância. O tipo especifica, em algum nível de abstração, que interface o cliente espera que o servidor implemente. A instância especifica qual implementação em particular é desejada, entre as várias possíveis. Em determinados modelos de RPC além, de ter tipos de interfaces, são definidos também tipos de implementações diferentes para um mesmo tipo de interface, dos quais podem existir várias instâncias. Nestes casos é então necessário precisar o tipo da interface, o tipo da implementação e a instância desejada. A implementação de RPC não define a semântica do nome de uma interface, mas sim a forma pela qual é utilizado o nome para localizar o exportador.

Analisaremos a seguir algumas dessas formas, desde as mais

simples realizadas na compilação, até as mais complexas que permitem maior flexibilidade usando gerenciadores de servidores que tomam conta de várias instâncias de forma dinâmica.

A forma mas simples é aquela na qual o endereço do servidor na rede pode ser compilado no código do cliente pelo programador: ele pode ser encontrado numa tabela que contenha os nomes e endereços dos servidores, ou através da solicitação da informação a um operador. Estes métodos não são muito flexíveis mas são adequados em alguns casos, como por exemplo, em aplicações cuja configuração é estática, e nas quais o custo extra causado pela ligação imediatamente anterior à execução da chamada afeta o desempenho. Isto pode acontecer no caso da comunicação de alarmes em aplicações de controle de processos.

Outra forma que apresenta uma maior flexibilidade é a ligação realizada depois da compilação da aplicação, quando esta é composta de módulos separados, mas antes da execução. Para isto é necessário ter um mecanismo, separado da linguagem de programação em pequena escala, que permita definir a configuração do sistema e as suas ligações, para ser carregado numa determinada arquitetura de hardware. Esta forma permite programar a aplicação de maneira independente da arquitetura na qual será executada. Analisaremos este caso com mais detalhe na apresentação de nossa proposta.

As formas mais flexíveis de ligação são as que podem ser realizadas em tempo de execução. Neste caso também existem vários modos de implementação que oferecem graus diferentes de flexibilidade. Em [19] são discutidas por BIRRELL e NELSON algumas das formas oferecidas na implementação do modelo de NELSON, no projeto CEDAR, que utiliza a base de dados distribuída Grapevine [18] no processo de ligação. Entre as mais flexíveis, esta implementação permite nomear uma instância de uma interface mas sem que esta esteja ligada a uma determinada máquina: a escolha da máquina exportadora será definida em tempo de execução. Outra forma mais flexível ainda, é aquela na qual o importador especifica somente o tipo da instância mas não a própria instância, deixando essa escolha para o suporte de execução da RPC. CEDAR também permite ao cliente instanciar interfaces e importá-las, de forma que ele possa se ligar a várias máquinas exportadoras dinamicamente, mesmo quando ele não

pode saber antecipadamente a quantas máquinas vai querer se ligar. Esta forma é útil para implementar alguns algoritmos que utilizam várias máquinas de forma não totalmente definida, como por exemplo no caso do gerenciador de transações atômicas distribuidas.

Existem algumas situações nas quais o cliente precisa mudar as suas ligações durante a execução. Por exemplo, isto pode acontecer no caso de uma chamada falhar, ou de forma deliberada, quando são atualizadas as cópias de dados replicados. Neste caso a ligação anterior pode ser escondida no cliente, de maneira que as ligações possam ser comutadas entre vários servidores similares ligados ao cliente, de forma bem eficiente. Outra situação possível é quando o servidor deseja alterar a ligação. Isto pode ser necessário no caso do serviço precisar ser levado para outra máquina, ou para permitir que uma nova versão do servidor seja instalada.

Já foi mencionado que podem existir vários servidores disponíveis para atender a uma chamada remota. Um cliente pode estar ligado a um número qualquer deles, um de cada vez, mas pode estar ligado também a mais de um simultaneamente. Logicamente uma ligação deste tipo cria uma comunicação de difusão ("multicast"), que implica que quando a chamada for feita, vários servidores a processarão. É apontado em [173] que é útil ter uma forma simples de comunicação de difusão, mas que não está claro como integrá-la na linguagem de alto nível. Por um lado porque o modelo é alheio á maioria das linguagens concorrentes, e por outro porque a manipulação dos resultados depende de quantas respostas são necessárias.

Para permitir maior flexibilidade ainda no projeto de aplicações que utilizam RPC, algumas implementações incluem um **serviço de ligação** que atua como agente entre os clientes e os serviços requeridos. Isto é, existe um **ligador** com funções bem especificadas que atua entre o cliente e o servidor. Para ilustrar estes conceitos podemos citar uso da base de dados distribuida Grapevine utilizada na implementação do modelo de NELSON, já mencionada anteriormente. A maioria dessas funções já foi mencionada nos casos analisados anteriormente, mas a sua discussão aparecia diluida na apresentação da implementação do mecanismo de RPC.

Os servidores se comunicam com o ligador para registrar as interfaces que eles exportam, e os clientes procuram a localização das interfaces importadas consultando o ligador. O acesso ao ligador é geralmente realizado através de chamadas remotas. Quanto maior for a sofisticação do ligador, mais flexíveis serão os serviços por ele oferecidos. Isto se relaciona com as formas vistas anteriormente de nomeação e identificação de instâncias de interfaces e com o caráter dinâmico das ligações.

Além dos requisitos funcionais, o ligador deve ser robusto em relação a falhas, isto é, ele deve ter tratamento de erros incluído; para seu bom desempenho não pode permitir ocorrer situações de gargalo. Uma maneira de satisfazer estes requerimentos é distribuir as funções do ligador entre vários servidores e replicar neles a informação. Infelizmente, para satisfazer todas as exigências mencionadas, é necessário um pacote de gerenciamento de base de dados muito complexo e, por isto, às vezes a funcionalidade oferecida pelo ligador é menor do que a esperada.

IV.3.1.6 GERÊNCIA E SEMÂNTICA DOS SERVIDORES

Embora estes dois aspectos estejam relacionados ao problema de ligação tratado na seção anterior, estão ainda mais relacionados entre si, e por isto serão analisados em conjunto nesta seção.

Em algumas implementações de RPC podem existir várias instâncias de um servidor, seja na mesma máquina, seja em máquinas diferentes, para prover balanceamento de carga ou certa elasticidade em relação a falhas. O mecanismo de ligação descrito anteriormente permite a escolha de uma instância arbitrária em determinados casos, quando o cliente tenta importar a interface apropriada. Quando a ligação permanece fixa durante toda a execução da aplicação podemos caracterizar os servidores de estáticos, e eles podem guardar os estados entre chamadas sucessivas de procedimentos. No caso de um servidor receber várias chamadas de diferentes clientes, ele pode intercalar essas chamadas e gerenciar concorrentemente vários conjuntos de

informações de estado.

Em outras implementações existe um **gerenciador de servidores** que cria servidores sob demanda. Por exemplo, descreveremos a seguir uma maneira de funcionar. O cliente contacta o ligador na forma comum, mas ele obtém de volta o endereço do gerenciador de servidor. O cliente contacta a seguir este gerenciador, que devolverá para ele o endereço do servidor do tipo requerido para uso privado. Em determinados ambientes, o gerenciador de servidor atua como um gerenciador de recursos e escolhe um servidor ocioso a partir de um conjunto de servidores criados anteriormente; em outros ambientes um novo servidor é criado sob demanda. Em ambos os casos o cliente possui uso exclusivo do servidor durante o intervalo de tempo que dure a transação ou sessão. Em outros casos, o próprio gerenciador de servidor pode criar uma variedade de servidores em resposta a chamadas adequadas de primitivas de criação de servidores, e ele é denominado de **servidor genérico** [173].

Outra estratégia de gerenciamento de servidor menos comum é aquela, na qual cada chamada do servidor cria uma nova instância do servidor que deixa de existir quando a chamada for completada. Ela pode ser denominada de estratégia **instância-por-chamada** [173].

O servidor estático representa a forma mais rudimentar, e precisa que o implementador gerencie os estados de forma concorrente dentro do servidor. É adequado para serviços padrões mas é difícil fazer balanceamento de carga de instâncias diferentes. Com os gerenciadores de servidores, cada servidor normalmente atende um único cliente, de maneira que a necessidade de balanceamento de carga desapareça, e o código que implementa o servidor precise gerenciar um único conjunto de estados. No último caso, não é preciso guardar a informação de estado entre chamadas.

Nesta primeira parte, nos concentramos nas formas de gerenciamento dos servidores. Outro problema a ser analisado é a sua semântica. Para isto lembremos como é executada a RPC para estudar como ela será atendida.

Em geral a escolha de onde o procedimento remoto será executado é adiada até o tempo de execução, de maneira que, com exceção dos sistemas embutidos ("embedded systems"), o código do

servidor será identificado e carregado na máquina apropriada ao mesmo tempo que o cliente. Mais frequentemente o servidor é instalado antes que o cliente execute, ou é criado (implicitamente ou explicitamente) quando o cliente precisar dele.

Analisaremos agora como as várias formas de implementar RPC escolhem diferentes mecanismos para criar os servidores; isto resulta em diferentes semânticas para os servidores.

Como já foi mencionado antes, uma estratégia possível é a de ter um servidor que existe somente durante o intervalo de tempo que demora a execução da chamada. Neste caso ele é criado pelo suporte de execução de RPC somente quando chegar uma chamada. Depois dela ter sido atendida, o servidor é destruído. Este esquema implica num custo grande, e portanto só é adequado para servidores que tenham efeitos colaterais interessantes, como por exemplo ler ou escrever um arquivo, ou fazer uma outra chamada remota aninhada. Somente em casos simples o servidor pode ser criado antes da chamada, já que não se sabe que procedimentos serão chamados, e também porque o servidor não pode esconder na memória dados importantes entre chamadas. Qualquer informação de estado que precisar ser guardada entre várias chamadas, deverá ser entregue ao sistema operacional que dá suporte ao sistema. Para evitar esta sobrecarga o programador deverá projetar a interface com o servidor, de maneira a passar a informação de estado para e desde o servidor em cada chamada, perdendo desta forma a abstração de dados através da interface cliente/servidor. Como resultado o mecanismo de RPC perde parte do seu interesse para ser utilizado pelo programador.

Outra forma, diametralmente oposta, é a de ter um servidor permanente que existe durante toda a execução do sistema, geralmente compartilhado por vários clientes. Um servidor deste tipo pode guardar informação de estado na memória entre as chamadas, e pode representar uma forma mais clara e mais abstrata de interface para seus clientes. Se o servidor é compartilhado entre vários clientes, os procedimentos remotos que ele oferece devem ser projetados de maneira que pedidos intercalados ou concorrentes dos diferentes clientes não interfiram ente si. Aqui podemos distinguir duas formas de atendimentos das chamadas dos diferentes clientes, que são ilustradas pela Figura (IV.11) de

[119]

As chamadas de diferentes clientes para o procedimento remoto do servidor podem ser atendidas de **forma concorrente** associando um processo a cada chamada. O suporte em tempo de execução da RPC pode associar um número determinado de processos ociosos para atender às chamadas, e ir alocando cada um à medida que as chamadas cheguem; desta forma é colocado um limite de concorrência prefixado. Outra implementação é criar um processo para atender cada chamada e destruí-lo quando acabar a sua execução. Esta forma implica num custo maior, mas garante que serão criados processos para todas as chamadas sem correr o risco de bloqueá-las, como pode acontecer no outro caso se o número de processos determinado é inferior ao número de chamadas recebidas. Deve-se tomar cuidado neste esquema no caso do procedimento P da Figura (IV.11) gerenciar dados compartilhados; nessa situação, M deve ser um monitor que sincronize as chamadas de P (e de outros procedimentos de M), ou deve-se utilizar qualquer outro esquema de controle de acesso concorrente aos dados compartilhados.

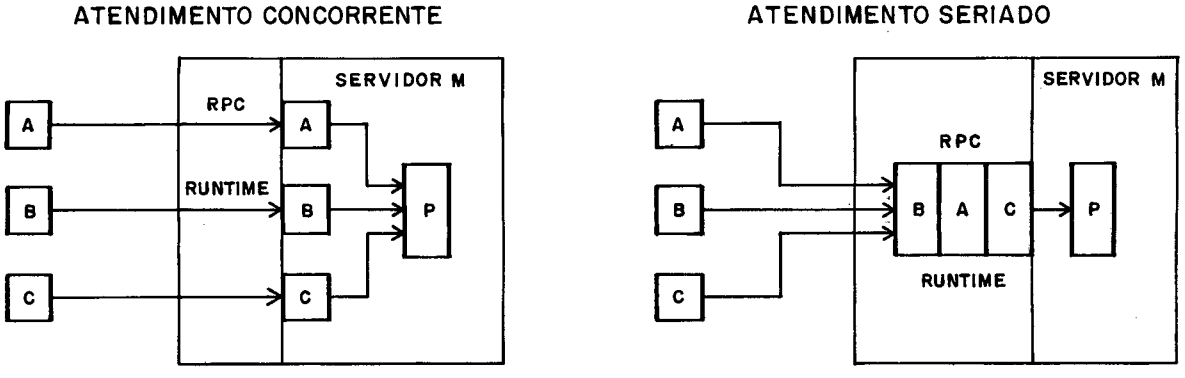

FIGURA IV.11 - POLÍTICAS DE ATENDIMENTO DE RPC

Uma outra forma de atendimento das chamadas é a **serial**, onde existe um único processo associado ao servidor que atenderá as chamadas uma de cada vez, depois destas terem sido recebidas e enfileiradas pelo suporte em tempo de execução de RPC. A vantagem deste esquema é que P é automaticamente sincronizado de forma análoga a uma entrada de procedimento de monitor ("entry procedure"). Por outro lado a desvantagem reside na possível ocorrência de bloqueio perpétuo. Por exemplo, isto poderia acontecer no caso de C esperar por uma variável de condição que seria sinalizada por A ou B se eles fossem executados de forma concorrente. Outro exemplo, não ligado a problemas de sincronização, é o caso de C chamar P e P chamar C de forma recursiva; a chamada de P para C seria enfileirada esperando ser completada a chamada de C para P.

Outras formas intermediárias de atendimento de chamadas podem ser implementadas combinando aspectos de uma e da outra, e elas serão mais ou menos adequadas dependendo das aplicações. Porém de um modo geral a forma mais utilizada é a do atendimento concorrente por ser a mais homogênea e a mais coerente com o esquema de RPC.

Finalmente podemos mencionar uma forma de comportamento intermediário de implementação de servidores que consiste em combinar o uso de servidores permanentes com o servidor genérico mencionado anteriormente. O servidor genérico devolve um identificador único para referenciar o novo servidor que será particular do cliente, já que ele é o único que o possui. O servidor pode manter informação de estado entre chamadas mas não pode compartilhar seus dados com outros clientes. Outra forma intermediária é a de usar o servidor genérico para implementar o servidor com a semântica de criação por chamada descrita anteriormente, de forma que seja criado um novo servidor cada vez que é realizada uma chamada.

IV.3.1.7 TRANSPARÊNCIA

Como já citamos anteriormente, para o modelo de RPC embutido na linguagem de alto nível existem duas formas distintas: a **RPC transparente** e a **RPC não transparente**. Para orientar a discussão

sobre as duas formas escolhemos o trabalho de NELSON [119] que descreve o projeto para uma implementação de RPC transparente a ser embutido na linguagem Mesa, e o trabalho de HAMILTON [64] que descreve a implementação de RPC não transparente embutido na linguagem CLU concorrente. Estes trabalhos foram escolhidos por serem os primeiros a tratarem este assunto em profundidade, tendo norteado a maioria das implementações posteriores.

NELSON define o conceito de transparência da seguinte maneira: "dois mecanismos são transparentes se eles têm sintaxe e semântica idênticas. Em particular, um mecanismo de RPC transparente, no nível da linguagem, é aquele no qual os procedimentos locais e remotos são efetivamente indistinguíveis para o programador".

Depois de comparar os mecanismos de comunicação remota entre processos, os baseados em mensagens remotas e os baseados em procedimentos remotos, o autor conclui que o mecanismo de RPC é frequentemente mais fácil de ser integrado numa linguagem de alto nível, uma vez que a maioria das linguagens de programação é procedimental. Por outro lado, para os projetistas e programadores de sistemas, familiarizados com o estilo de programação das linguagens convencionais, é mais fácil a transição para programação em sistemas distribuídos, se estes dispuseram de uma primitiva para comunicação distribuída que preserve as noções importantes de abstração de dados. RPC é uma primitiva que possui todas as propriedades desejáveis, inclusive a semelhança da chamada local de procedimento.

O objetivo de NELSON foi implementar, no projeto de ambiente de programação CEDAR [162, 51], o mecanismo de RPC totalmente transparente embora reconhecendo que o seu desempenho era pior que das chamadas locais. Sua implementação é baseada na geração automática de "stubs", a partir das especificações procedimentais das interfaces. As vantagens deste enfoque consistem no fato de não precisar modificar o compilador da linguagem utilizada, além de tornar as chamadas remotas indistinguíveis das locais. Entretanto não foi conseguida uma equivalência total na semântica das chamadas, e este é o maior problema na integração de RPC nas linguagens. Já foram mencionadas as semânticas de chamada escolhidas com o propósito de se obter a maior similaridade possível entre chamadas locais e chamadas remotas. A

implementação de BIRRELL e NELSON [19] colocou menos ênfase na transparência da semântica da chamada em relação à projetada no sistema Emissary de NELSON [119].

Outro aspecto ligado à semântica da chamada, requisito essencial para a sua transparência, é poder embutir no nível da implementação de RPC o tratamento de erros. Infelizmente isto também não foi possível implementar totalmente, e erros persistentes na comunicação com servidores são transmitidos aos clientes, levantando exceções.

Por outro lado, outra vantagem salientada no modelo transparente é a independência em relação à configuração do sistema, que permite separar totalmente a programação em pequena escala dos componentes do programa, da arquitetura de hardware específica na qual será executada. A configuração pode ser realizada através de primitivas específicas, ou através de outra linguagem apropriada para programação em larga escala. No entanto, na implementação já citada este enfoque não foi seguido fielmente e as ligações ficaram sob controle explícito do programa.

Nem o Emissary, nem o sistema CEDAR conseguem dissimular outra diferença importante entre chamadas locais e remotas, que é o desempenho bem menor neste último caso.

Devido a esta série de problemas, HAMILTON defende o uso de RPC não transparente. Ele acha que, por exemplo em relação ao problema do desempenho, o implementador deve estar ciente de quais chamadas são locais e quais são remotas, para projetar um sistema o mais eficiente possível. Um dos objetivos do sistema CEDAR é permitir que um programa seja escrito e testado para depois ser fragmentado, sem precisar mudanças no código, a fim de ser distribuído entre vários processadores. Entretanto HAMILTON defende que, os custos no desempenho, e na semântica das chamadas remotas, fazem com que as unidades de distribuição e a natureza das suas interconexões precisem ser consideradas como decisões fundamentais, a serem tomadas logo no projeto do sistema, e não durante sua implementação. O autor considera que o programador deve controlar a alocação dos componentes do sistema no hardware específico, a fim de decidir quais chamadas de procedimento podem ser locais e quais remotas em função de um melhor desempenho do sistema.

Portanto, um enfoque alternativo é reconhecer a inevitável falta de transparência, e prover ao programador da aplicação a oportunidade de explorar a melhor distribuição de seus componentes e o melhor desempenho de protocolos mais simples. Por outro lado HAMILTON [64] defende também o tratamento explícito dos erros, em lugar do tratamento embutido na implementação de RPC, por achar que em vários casos a melhor resposta para um erro pode depender da aplicação específica.

Podemos concluir que existem vantagens e desvantagens nos dois esquemas, e se por um lado os principais objetivos de RPC transparente são a simplicidade e transportabilidade, por outro lado o RPC não transparente oferece maior flexibilidade e melhor desempenho.

IV.3.1.8 CONSISTÊNCIA DE TIPO

Na construção de um sistema distribuido, é comum que os vários processos co-operantes que o constituem sejam desenvolvidos separadamente. Pelos conceitos básicos de engenharia de software, é necessário que as interfaces entre os processos estejam definidos sem ambiguidade para garantir a compatibilidade dos componentes finais. Este requisito coincide com o do mecanismo de RPC, no qual é necessário que constem na interface, a ordem e o tipo dos parâmetros e resultados de cada procedimento. A implementação pode usar esta informação para prover certa segurança tanto ao servidor quanto ao cliente, de maneira que uma ligação só seja permitida quando for garantido que ela foi feita por um cliente usando a mesma definição de interface que o servidor.

HAMILTON ressalta em [64], que na sua implementação de RPC foi decidido prover interfaces no nível da linguagem que possam ser testadas em tempo de compilação, e cujos tipos possam ser automaticamente codificados e decodificados a partir de áreas de memória ("buffers") pelo suporte em tempo de execução da linguagem. Mas ele reconhece que isto não garante que não possam acontecer problemas de incompatibilidade em tempo de execução. Num sistema distribuido, onde ocorrem modificações frequentemente, um módulo pode ter sido modificado e recompilado

com uma interface diferente. Os módulos que inicialmente tinham sido testados na primeira compilação não são mais compatíveis depois de recompilados. Devido a este problema foi decidido acrescentar verificação de consistência também em tempo de execução, para garantir que as interfaces remotas concordem com o tipo esperado, e desta maneira oferecer um mecanismo não só com teste de tipo mas também seguro.

Podemos notar que, em relação à forma de implementar o teste de tipo, existem conflitos entre flexibilidade e eficiência. Por um lado pode ser carregada em cada máquina a tabela de símbolos de cada interface exportadora. Isto permite fazer o teste completo em tempo de execução, mas às custas de espaço na memória para a tabela, e de tempo gasto para fazer o teste. Por outro lado, existe a forma de implementação utilizada em MESA, que gera um identificador único para cada interface indicando quando foi compilada. A linguagem exige que todos seus módulos importadores sejam compilados com o mesmo identificador de interface para garantir a sua compatibilidade.

O número de formas possíveis de verificar a compatibilidade de tipos está relacionado com o adiamento do momento de ligação discutido anteriormente. Quanto mais for adiado o momento de ligação, mais aumenta a generalidade do sistema mas às custas de tornar complexo o teste. As ligações feitas no início, por outro lado, reduzem significativamente a generalidade mas também o custo da ligação. Em MESA o teste de tipo das interfaces é feito durante a ligação, verificando se os identificadores das interfaces são iguais. Este esquema, entretanto, apresenta a desvantagem de que, no caso de ser recompilada uma definição de interface, todos os programas que usam essa interface precisam ser recompilados também, e é necessário refazer as ligações.

Existe outro aspecto, ligado parcialmente a este problema, que aparece em ambientes heterogêneos, tanto de software quanto de hardware. Nestes casos é necessário fazer traduções de tipos de forma segura para garantir o sucesso do mecanismo de RPC. Já foi mencionado anteriormente que, nestas situações, o ideal é definir linguagens intermediárias para especificação de interfaces.

IV.3.1.9 CONSIDERAÇÕES SOBRE EFICIÊNCIA

O uso de RPC é frequentemente utilizado para distribuir uma aplicação entre várias máquinas de forma de ter vários servidores compartilhados por um conjunto de clientes. Claramente RPC provê um mecanismo adequado para intercomunicação entre processos para sistemas estruturados dessa maneira. Embora este mecanismo apresente as vantagens de transparência e de similaridade de uso com as linguagens procedimentais, é necessário avaliar a sua eficiência.

Podemos identificar na execução de uma RPC duas formas de atraso. Consideremos um servidor simples que não faz chamadas a outros servidores. Pode acontecer que para executar uma chamada, ele precise ter acesso a um recurso que esteja temporariamente ocupado; neste caso ele ficará bloqueado até que o recurso seja liberado e esta espera é chamada **atraso local**. Este atraso degrada o desempenho do sistema se o servidor não puder atender outras chamadas durante esse intervalo de tempo. A segunda forma de atraso, denominada **atraso remoto**, ocorre quando o servidor chama uma função remota que envolve um tempo considerável de computação ou envolve um tempo de transmissão significativo. Outro caso que provoca também este atraso é quando a função remota pertence a um outro servidor, e acontece a situação descrita acima para ilustrar o atraso local. Boas implementações de RPC devem prover mecanismos que permitam construir um sistema eficiente apesar destes atrasos, e isto implica que o servidor deva poder atender pedidos múltiplos simultaneamente, ou assinalar uma exceção para o cliente. Desta forma o cliente poderá chamar outro servidor ou iniciar alguma outra atividade.

LISKOV et alii [100] analisaram as diferentes combinações de mecanismos de comunicação e de estruturas de processos para escolher a forma mais poderosa para contornar o problema das demoras locais e remotas. Considerando os mecanismos de comunicação síncronos (RPC) e assíncronos (troca de mensagens) e estruturas estáticas e dinâmicas de processos, concluiram nesse trabalho, que a combinação que oferecia o maior poder de expressão é o uso de RPC com estrutura dinâmica de processos.

Esta é a que permite resolver os problemas levantados, da forma mais fácil e mais eficiente. Entretanto, o uso de processos providos pelo sistema operacional pode diminuir significativamente o tempo de resposta do servidor por causa do tempo necessário para a sua criação. Como o sistema operacional geralmente deve dar suporte à execução de várias linguagens, na criação de processos, ele precisa definir um contexto a ser salvo, que seja o mais geral possível para poder satisfazer os requisitos de todas as linguagens. Desta forma o custo de transferência de controle entre processos pode ser alto, devido à grande quantidade de informação a ser salva e restaurada. Por causa deste problema, é sugerido dar suporte a processos ou tarefas "baratos", cuja criação consome menos tempo, ou ao nível do compilador da linguagem, ou através de um pacote que crie processos no nível da linguagem como complemento. Estes processos são mais baratos, porque para cada linguagem pode ser definido o contexto específico a ser salvo na transferência de controle, que será geralmente menor do mencionado no caso anterior. Mas existem outras vantagens ainda que são, por um lado o fato dos processos não poderem ser interrompidos, e por outro, que seu segmento de código e espaço de endereçamento podem ser compartilhados por todos os outros processos ou tarefas.

Estes processos provêem também uma maneira de contornar uma restrição importante de RPC, que é a possível diminuição de paralelismo. Lembremos que o processo cliente, quando faz uma chamada remota, fica bloqueado e ocioso esperando a resposta. O cliente e o servidor atuam de forma análoga às co-rotinas. Embora uma das vantagens de RPC seja a sua propriedade de sincronização, várias aplicações distribuídas podem se beneficiar tendo acesso concorrente e paralelo a servidores múltiplos. Processos em clientes podem prover um mecanismo adequado, se o endereçamento no protocolo subjacente é suficientemente poderoso, para tomar conta corretamente do roteamento das respostas.

Uma alternativa para a solução que acabamos de descrever é o uso de **resposta prematura** ("early reply"), ilustrada na Figura (IV.12):

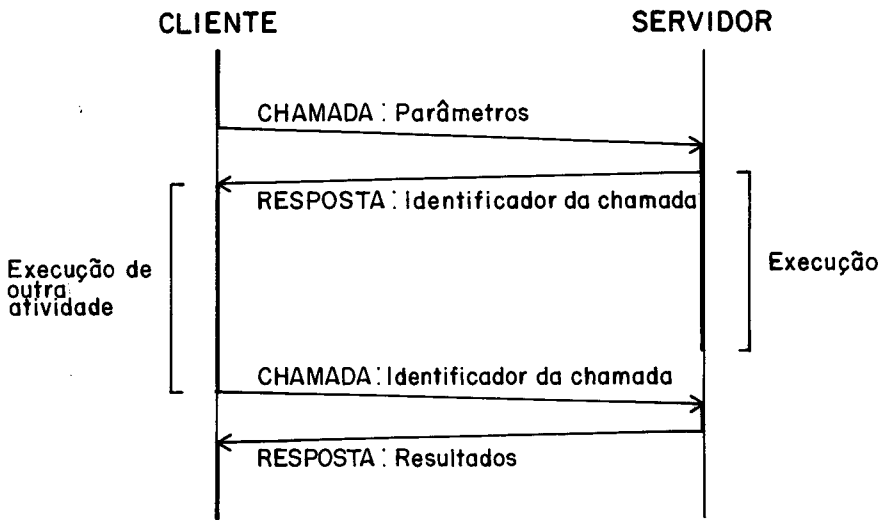

FIGURA IV.12 - RESPOSTA PREMATURA

Neste esquema a chamada é dividida em duas RPC separadas, uma passando os parâmetros ao servidor, e a outra requerendo os resultados. O resultado da primeira chamada pode ser um identificador da chamada ("tag"), que será passado de volta na segunda chamada para associar as duas, e identificar os resultados corretos. Em princípio o cliente pode definir um intervalo entre as duas chamadas, que pode ser aproveitado para executar algum outro procedimento independente do resultado da chamada, e pode até executar outras RPC. Entretanto, se o pedido dos resultados for adiado muito, isto pode causar congestionamento ou demora não necessária no servidor.

Outra variante é a proposta de GIMSON [60] baseada no uso de "buffers", ilustrada pela Figura (IV.13). É criado um servidor de "buffers" de maneira que, quando é feita uma chamada remota para um serviço X, ela se traduz numa chamada ao servidor de "buffers" que guardará os parâmetros junto com os nomes do serviço e do cliente. Depois o cliente fará chamadas periódicas ao servidor de "buffers" para ver se a chamada foi executada, quando recolherá os resultados. Entretanto, os servidores também consultarão o servidor de "buffers" para comprovar se existem chamadas

pendentes. Nesse caso são apanhados os parâmetros, a chamada é executada, e através de outra chamada ao servidor de "buffers" são entregues os resultados que serão armazenados até o cliente correspondente recolhê-los.

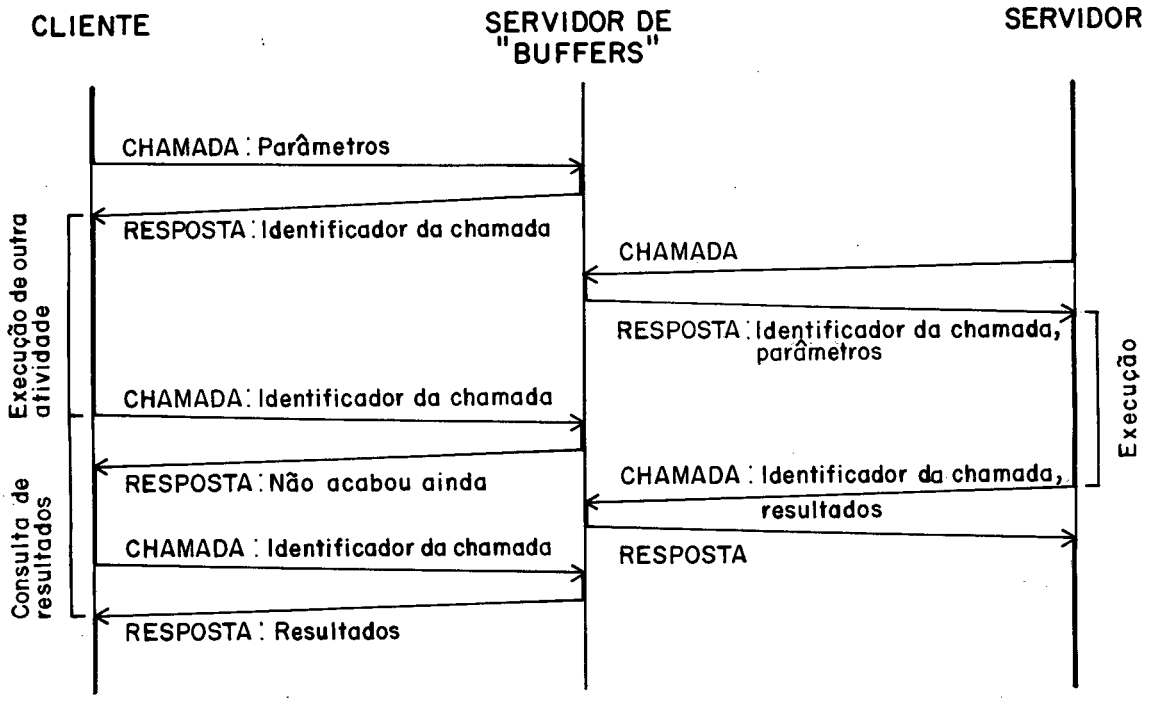

FIGURA IV.13 - CHAMADA ATRAVÉS DE UM SERVIDOR DE "BUFFERS"

Considerando os vários exemplos ligados ao suporte de processos para RPC, podemos concluir que o ideal é poder prover um mecanismo barato de criação dinâmica de processos, como complemento da RPC. Esta é a forma mais poderosa e expressiva para contornar algumas das limitações de eficiência da RPC em ambientes baseados em servidores. As outras alternativas apresentadas podem diminuir o problema, mas forçam o usuário a adotar uma forma de programação, que, além de ser menos elegante, perturba o estilo convencional.

IV.3.3 ALGUNS PROBLEMAS AINDA NÃO RESOLVIDOS

É nosso objetivo nesta seção apontar alguns problemas levantados por autores que adquiriram alguma experiência no uso de RPC, e para os quais foram achadas soluções parciais, nem sempre satisfatórias, ou para os quais ainda não foram vislumbradas soluções.

Em particular, TANENBAUM e RENESSE [164] têm separado estes problemas em quatro categorias que incluem problemas conceituais com o modelo, problemas técnicos em relação à transparência do mecanismo, problemas em relação à ocorrência e ao tratamento de falhas, e, por último, problemas de desempenho.

Os problemas conceituais se referem ao fato que, segundo eles, o mecanismo de RPC é especialmente apropriado para sistemas distribuídos estruturados em comunicação entre clientes e servidores. Mas esta é simplesmente uma estrutura particular dentro da variedade de modelos de estruturar sistemas distribuídos. Existem exemplos de aplicações nos quais é difícil estruturar o sistema na forma única de clientes e servidores. É apresentado um exemplo de um "pipeline" de UNIX no qual não é claro distinguir qual é o processo cliente e qual é o processo servidor. Em determinados casos um processo pode ao mesmo tempo funcionar como cliente e servidor. Como exemplo é citado o caso de um conjunto de estações de trabalho que compartilham um servidor de arquivos, mantendo arquivos locais em cada estação. No caso de ser feita uma modificação num arquivo local, esta deverá ser informada ao servidor de arquivo para modificar as outras cópias locais, e para isto o servidor assumirá funções de cliente, mas os outros clientes não ficam sabendo que deverão se comportar como servidores. Isto força a estruturação de sistemas segundo o modelo cliente/servidor sendo que para determinadas aplicações esta forma pode não ser a mais adequada.

Outro problema colocado é o fato dos servidores terem uma única linha de controle para atender os pedidos. Para contornar esta situação, o programador é forçado a escrever servidores com linhas múltiplas de controle, para aumentar a sua eficiência, através de uma multiprogramação interna.

Um último problema, apontado também em [173], é a necessidade de implementar, na maioria de sistemas de redes

locais, comunicação para várias máquinas com mensagens de tipo difusão ("multicasting"), tanto para chamadas como para respostas. Ainda não se tem explorado muito o poder e as dificuldades do mecanismo de RPC para incluir este tipo de comunicação, apesar de algumas soluções terem aparecido como as de BANATRE et alii em [14] e CHERITON e DEERING em [39].

Em relação aos problemas de transparência as principais dificuldades levantadas se relacionam com as limitações provocadas pela proibição do uso de ponteiros de parâmetros passados por referência, e de procedimentos passados como parâmetros, como já foi mencionado numa seção anterior, o que diferencia as chamadas locais das remotas. Esta situação limita os programadores que utilizam o mecanismo de RPC a usar um subconjunto da linguagem. Este problema se estende também ao uso de variáveis globais, oferecidas pela maioria das linguagens, e que são utilizadas por chamadas locais, mas que numa aplicação distribuida não fazem sentido. Mesmo para sistemas centralizados, as linguagens mais modernas usam o conceito de módulo, ou outro análogo, para particionar o espaço das variáveis globais.

Algumas soluções apontadas são muito caras ou inviabilizam o princípio de transparência, portanto não resolvem o problema. Segundo TANENBAUM e RENESSE o mecanismo de RPC deve ser totalmente transparente para proteger o programador. Com linguagens convencionais estruturadas em blocos não é claro que isto seja possível. Como já foi mencionado anteriormente, HAMILTON [64] defende que os aspectos não transparentes devem ser explícitos e não escondidos. Outra forma de atacar o problema é projetar linguagens de programação novas, que não contenham ponteiros, parâmetros passados por referência, variáveis globais ou outros aspectos que causam problemas para RPC. Por exemplo, com linguagens de programação funcionais é possível obter transparência total.

Uma das diferenças fundamentais entre uma chamada local e uma remota, consiste na enorme variedade de erros que podem acontecer durante a execução desta última. Um dos objetivos de distribuir um sistema é incrementar a tolerância a falhas, portanto o uso de RPC nem sempre é o mais atrativo.

Na implementação de RPC, se o erro não for tratado de forma transparente, o que nem sempre é possível, deve ser passada a

ocorrência do erro ao cliente, que deverá então estar preparado para tratá-lo. Isto evidentemente faz com que o código de uma RPC seja diferente de uma chamada local, violando novamente a transparência.

Uma das formas, já mencionada, utilizada para implementar a semântica exatamente-uma-vez, para manter o princípio de transparência entre chamadas remotas e locais, exige que os procedimentos remotos sejam idempotentes. Acontece que nem sempre é possível programar operações idempotentes, como por exemplo no caso de operações de E/S, ou operações que acrescentam registros em arquivos. Uma forma de contornar este problema é controlar que uma operação não seja executada mais de uma vez, determinando um intervalo de tempo após o qual será comunicada a falha, ao invés de repetir a operação. O perigo aqui é que o servidor tenha entrado em colapso antes de ter executado a operação. Isto demonstra que, para operações não idempotentes, não há como obter uma mesma semântica para os dois tipos de chamadas, remota e local.

Outro problema é o fato de existir a possibilidade do servidor conter mais informação de estado sobre o cliente. Isto pode ocasionar sérios problemas no caso do servidor entrar em colapso entre RPCs e se reinicializar antes da execução da próxima RPC. Um servidor de arquivos, por exemplo, pode conter informação sobre arquivos abertos, posições atuais do arquivo, etc., que serão perdidas quando ele for reinicializado. Quando o cliente posteriormente quiser ler um determinado arquivo, o servidor não saberá mais que arquivo deve ler e qual é o estado atual dos arquivos.

Até agora temos analisado situações de colapso dos servidores, mas os colapsos dos clientes também ocasionam problemas sérios. Quando um cliente entra em colapso enquanto o servidor está ainda executando a chamada feita por ele, essa execução se transforma em órfã, como já foi visto anteriormente. Esses órfãos precisam ser eliminados, e segundo a opinião de TANENBAUM e RENESSE [164], nenhum dos métodos propostos até agora [148, 123] é elegante. Mais grave ainda é o fato de que alguns dos métodos utilizados para detectar e matar os órfãos correm o risco de matar um órfão no meio da execução de uma região crítica, podendo bloquear recursos importantes.

É apontado em [164] que, como as pastilhas ("chips") modernas de microprocessadores e as redes locais são altamente confiáveis, na prática não ocorrem muitos problemas. Entretanto, a confiabilidade das aplicações distribuídas é o produto da confiabilidade dos nós cooperativos. Portanto, em lugar de aumentar a confiabilidade, o que é um dos requisitos fundamentais dos sistemas distribuídos, o mecanismo de RPC pode aumentar a ocorrência de falhas.

Os autores do projeto Mayflower [65] colocam, em relação a este tema, que consideram fundamental, como passo futuro em seu projeto, acrescentar facilidades gerais para dar suporte a transações atômicas ou outros mecanismos similares. Mas por outro lado, eles acham que sempre existirão aplicações com requisitos de confiabilidade particulares, que desejarão prover seus próprios mecanismos flexíveis de tratamento de falhas, em lugar de usar um mecanismo padrão. Outro ponto mencionado por eles é a dificuldade de depurar sistemas distribuídos que consistem de um conjunto de processos concorrentes em máquinas distintas. Os autores acham que esta é uma área que precisa ser estudada em profundidade, já que até agora não têm aparecido propostas em relação à definição de ferramentas para a depuração de tais sistemas. Em relação a seus passos futuros, eles estão planejando construir um mecanismo de RPC que seja independente de linguagem, já que cada dia mais são encontradas aplicações formadas por servidores e clientes escritos em linguagens distintas. Uma dúvida por eles apontada é como vai ser possível conciliar os requisitos de eficiência e controle de tipos com a independência de linguagem.

Finalmente gostaríamos de citar algumas opiniões de BACARISE e WILBUR [10], que consideram que o mecanismo de RPC representa uma técnica valiosa que permite construir uma variedade grande de aplicações distribuídas. Entretanto eles o consideram um simples mecanismo de transporte, e acham que em seu redor devem ser construídas ferramentas para implementar gerenciadores de servidores. Características de ligação e escopo de nomeação precisam ainda de estudo considerável, e devem ser integrados novos conceitos em relação à segurança e autenticidade da rede.

IV.4 APRESENTAÇÃO DA PROPOSTA DE RPC

IV.4.1 DESCRIÇÃO GERAL DO MECANISMO

Como já foi mencionado anteriormente, decidimos acrescentar á Modula-2 um mecanismo de comunicação e sincronização entre processos localizados em nós diferentes, para estender o uso desta linguagem para programação distribuida. O mecanismo escolhido é a chamada remota de procedimento transparente, para poder ser utilizado na programação em pequena escala de forma idêntica à chamada de procedimento local. Será utilizada para a sua implementação o modelo de "stubs" gerados automaticamente. Devido à decisão tomada de não modificar o compilador de Modula-2, não foi acrescentada nenhuma palavra reservada nem primitiva nova.

O programa distribuido será especificado através da linguagem de configuração estática, já mencionada na introdução e que será apresentada no Capítulo VI. O programa será composto por módulos escritos em Modula-2, que, depois de compilados, serão utilizados pelo gerador de "stubs" para criar os "stubs" correspondentes aos módulos exportadores ("stub" do servidor e "stub" do cliente). A seguir, será processada a configuração através de um interpretador, que gerará os componentes a serem carregados em cada nó físico e será feito o carregamento necessário. Tanto o gerador de "stubs" quanto o interpretador da linguagem de configuração estão escritos em Modula-2. Existem ainda outros módulos que assistirão o programa na fase de execução, como o PCRP e um módulo de configuração utilizado pelo PCRP para ativar os procedimentos chamados.

Neste capítulo já foi apresentado o PCRP e será descrito a seguir o gerador de "stubs", enquanto que a linguagem de configuração, seu interpretador e o módulo de configuração serão apresentados no Capítulo IV, junto com um diagrama do ambiente como um todo ilustrado pela Figura (IV.25).

Para a implementação de chamadas remotas de procedimentos em sistemas distribuidos, foi escolhido o modelo que usa o conceito de "stub" (BIRRELL e NELSON [19], NELSON [119]) que foi utilizado como base para a nossa implementação que apresentaremos a seguir. Ilustramos o caso de um módulo (cliente) fazer uma

chamada de um procedimento contido num módulo remoto (servidor). Quando é executada a chamada remota, são envolvidos seis componentes de programa: o cliente, o "stub" do cliente, o suporte de execução de RPC (chamado "RPC runtime" e estando carregada uma cópia em cada nó), o módulo de configuração, o "stub" do servidor, e o servidor. A relação entre eles é ilustrada pela Figura (IV.14).

Para que uma chamada remota pareça formalmente igual a uma chamada local (propriedade de transparência já mencionada anteriormente), o cliente passa o controle para seu "stub" através de uma chamada local, ficando bloqueado até voltar o controle, depois da execução da chamada remota. O procedimento do servidor é ativado também através de uma chamada local feita por seu "stub". A descrição que segue explica a interrelação entre os dois "stubs".

O "stub" do cliente é o responsável pelo gerenciamento da chamada remota de um procedimento, feita pelo cliente, empacotando o nome do procedimento junto com os argumentos da chamada numa mensagem que é enviada para a máquina onde reside o servidor contendo o procedimento chamado. Ele é encarregado também de receber uma mensagem com os resultados da execução do procedimento chamado, e de passá-los para o cliente que ficou bloqueado esperando por eles.

O RPC runtime da estação do cliente ativado pelo "stub" do cliente é o responsável pelo envio da mensagem de chamada para a máquina que contém o procedimento, e pelo recebimento da mensagem de resultados da mesma máquina.

O RPC runtime da estação do servidor tem funções análogas ao da estação do cliente, com a diferença que é ele que ativa o "stub" do servidor através do módulo de configuração. O RPC runtime da estação do servidor recebe a mensagem de chamada que contém o nome do procedimento a ser executado junto com os parâmetros necessários para cada chamada. Ele passa esta informação para o "stub" do servidor, que é o encarregado da identificação do conteúdo da mensagem de chamada, e de passar a chamada do procedimento correspondente ao servidor que o contém, para iniciar a sua execução.

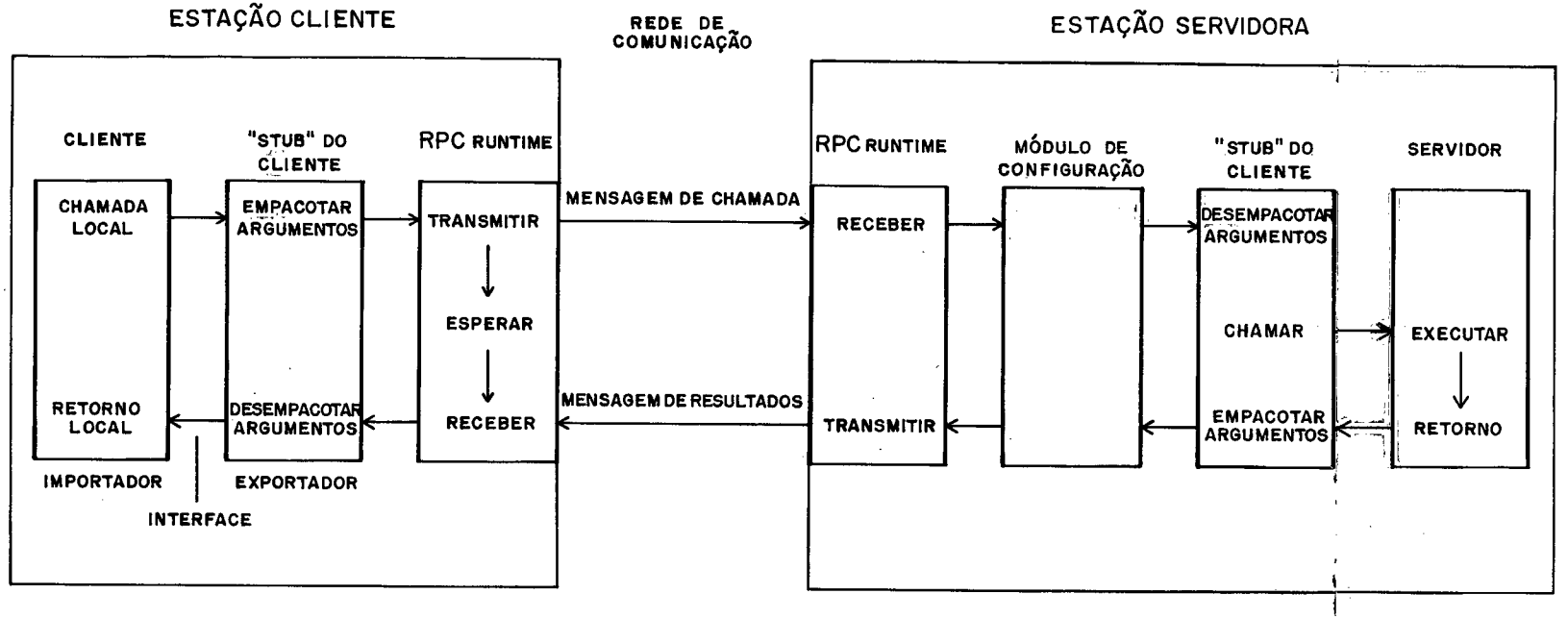

FIGURA IV.14 - ESQUEMA DA CHAMADA REMOTA DE PROCEDIMENTO

Quando a execução do procedimento acaba, o "stub" do servidor recebe os resultados, empacota-os e passa-os através do módulo de configuração, para o RPC runtime do servidor; este se encarrega de enviá-los para o RPC runtime da estação do cliente que fez a chamada.

A execução da chamada remota prossegue através da execução do "stub" do cliente que recebe os resultados. Ele desempacota os resultados para enviá-los ao cliente que pode então continuar a execução do seu código, após a chamada remota ter sido concluida.

Apesar do módulo de configuração estar presente em todas as estações, durante a execução de RPC só participa o da estação servidora. Por este motivo não colocamos o módulo de configuração na Figura (IV.14) na estação cliente. Este módulo será descrito detalhadamente no Capítulo VI.

Queremos salientar que o suporte de execução de RPC precisa conter todas as funções descritas, tanto para a estação servidora quanto para a estação do cliente, já que num mesmo nó físico podem estar localizados módulos clientes e módulos servidores. Mais ainda, podemos afirmar que nem sempre os módulos têm uma caracterização tão definida, e que um mesmo módulo pode exercer funções de cliente e servidor, isto é, pode chamar procedimentos de outros módulos e pode também exportar procedimentos próprios para serem chamados por outros módulos. Portanto será carregada uma cópia do suporte de execução de RPC em cada estação física.

Para cada chamada remota será necessário então criar os códigos dos "stubs" correspondentes, um para o cliente (isto é, para quem faz a chamada) e outro para o servidor (isto é, para quem contém o procedimento chamado). Este esquema será ilustrado com mais detalhe na Figura (IV.21) na seção IV.4.3.5.

Foi escolhida, para a implementação de chamadas remotas de procedimentos através de "stubs", o uso de um gerador automático que cria os "stubs", que são módulos codificados em Modula-2, tanto para o exportador (servidor) como para o importador (cliente). Em princípio, um "stub" de cliente implementa a mesma interface que o servidor que ele representa, e está totalmente especificado pelo módulo de definição do servidor. O gerador de "stubs" então processa este módulo de definição, e gera código para converter cada chamada a um procedimento da interface do servidor remoto num envio de mensagem à estação remota. Ao mesmo

tempo o gerador de "stubs" monta o código do "stub" do servidor, que receberá, através do suporte de execução de RPC, as mensagens enviadas pelo "stub" do cliente convertendo-as em chamadas locais aos procedimentos apropriados do servidor.

Em suma, a exportação de uma interface remota, detectada na configuração de um programa, resulta na geração de um par de "stubs" que comunicam entre si os detalhes da chamada remota e do retorno. Esta seria a implementação ideal, mas achamos mais simples outra solução alternativa na qual o gerador de "stubs" trabalha sobre o código obtido depois da compilação e antes do processamento da configuração. Desta forma o gerador de "stubs" gerará o par de "stubs", cada vez que ele detectar uma interface exportadora, independentemente de saber se ela é remota ou não. Depois de interpretar a configuração, para as exportações e importações remotas serão carregados os "stubs" correspondentes. Se por um lado esta solução implica numa sobrecarga para o gerador, ela facilita a reconfiguração, já que no caso de chamadas locais se transformarem em remotas, por causa de relocalização dos módulos nas estações físicas, os "stubs" já estarão gerados e será necessário somente fazer o seu carregamento. Em outras palavras, não é preciso chamar novamente o gerador de "stubs" e este processará a informação do sistema uma única vez, logo depois da compilação.

O processamento dos argumentos de uma chamada remota requer cuidado. Certos tipos de argumentos, por exemplo, ponteiros, não podem ser enviados para uma outra estação e tem que ser desreferenciados. No caso específico de parâmetros de saída (parâmetros VAR de Modula-2), somos obrigados a implementá-los usando a forma valor-resultado ("value-result"). Nesta forma, ao parâmetro corresponde uma variável local do procedimento que será inicializada com o valor do argumento na hora da chamada e cujo valor final na hora do retorno será atribuido ao argumento. Os "stubs" têm que converter a lista de argumentos para o texto da mensagem de chamada, e vice-versa. O gerador dos "stubs" fornece rotinas para empacotar/desempacotar argumentos, além de determinar se os argumentos devem ser transformados. Finalmente, o gerador de "stubs" detecta numa interface a existência de ítens inadmissíveis, como por exemplo, a exportação de variáveis.

Nas próximas seções serão analisados com detalhe as

implementações de suporte de execução de RPC (DA SILVA [45]) e do gerador de "stubs" (LAGES [88]), desenvolvidos em duas teses de mestrado.

IV.4.2 SUPORTE DE EXECUÇÃO DE RPC

Este suporte foi implementado por DA SILVA [45] e denominado por ele, protocolo de chamada remota de procedimento (PCRP).

O módulo PCRP foi implementado para trabalhar concorrentemente na modalidade de cliente e servidor, como já foi mencionado na seção anterior. Foram especificadas também as interfaces com a rede e com o chamado módulo de configuração, sendo este o responsável pela ligação implícita entre clientes e servidores. Descreveremos aqui as características principais da implementação e mais detalhes podem ser consultados na tese de DA SILVA [45].

IV.4.2.1 AMBIENTE

O sistema foi elaborado para rodar em microcomputadores IBM-PC/XT/AT ou compatíveis, usando MS-DOS 2.X ou compatível, com DOS 2.X interligados em rede local. Os programas foram escritos em Modula-2 e somente uma parte do código é dependente de arquitetura.

Em relação à confiabilidade da rede local quatro propriedades fundamentais são supostas:

i) se um pacote é transmitido inúmeras vezes, ele irá chegar a seu destino.

ii) pacotes corrompidos durante a transmissão são automaticamente descartados.

iii) os pacotes não são duplicados pela rede.

iv) os pacotes chegam na mesma ordem em que foram transmitidos. Portanto, falhas na rede acarretarão unicamente a perda de pacotes.

IV.4.2.2 PROTOCOLO DE COMUNICAÇÃO

O protocolo de comunicação implementa a semântica denominada **uma-e-somente-uma-vez**, na ausência de queda. Entretanto é sensível à queda e eventual recuperação tanto do cliente como do servidor.

O protocolo é baseado na transmissão de três tipos de pacotes: chamada, retorno e término, com a estrutura mostrada pela Figura (IV.15).

FIGURA IV.15 - FORMATO DOS PACOTES UTILIZADOS NA EXECUÇÃO DE RPC

Cada chamada, na sua forma mais simples, envolve a transmissão de dois ou três pacotes. O cliente envia um pacote de chamada, que é respondido pelo servidor com um pacote de retorno. O cliente envia a seguir um pacote de término, ou executa uma nova chamada para o mesmo servidor, que servirá também como indicação de término da chamada anterior.

Tanto o cliente quanto o servidor retransmitirão a chamada ou o pacote de retorno, de tempos em tempos, caso demore em chegar o pacote de retorno ou o de término ou nova chamada, respectivamente.

Cada pacote contém um identificador de chamada que está composto pelo identificador da estação cliente, o identificador do processo cliente e um identificador de transação; estes dois últimos definem uma atividade (BIRRELL e NELSON [128]) que é a propriedade de apenas executar uma RPC de cada vez.

O servidor mantém uma tabela contendo o registro de todos os identificadores das chamadas em andamento.

Com esta estrutura, é então possível identificar no caso

de retransmissão, se é uma nova chamada, ou uma chamada ou pacote de término duplicado que será descartado.

IV.4.2.3 CONCORRÊNCIA

Em relação aos modelos de atendimento das RPC já analisados, foi escolhido o tratamento concorrente das chamadas através de um conjunto fixo de processos existentes a partir da inicialização do sistema. Para a implementação destes processos é utilizado o núcleo de multiprogramação, analisado anteriormente, que oferece primitivas de criação/destruição de processos, sinalização para sincronização e primitivas de apoio para a geração de identificadores, sinalização e propagação de erros.

Como o núcleo não oferece primitivas para comunicação entre processos, o PCRP define um conjunto de endereços lógicos denominados portas, para prover meios de endereçamento e encaminhamento de pacotes entre processos. Em cada estação existe um conjunto de portas de dois tipos, portas-clientes e portas-servidoras, que serão gerenciadas através de monitores.

Na estação do cliente, para cada chamada remota é alocada dinamicamente uma porta-cliente que é liberada no término da operação. Os nomes da porta e da estação cliente são colocados no pacote de chamada para identificar o endereço de destino do pacote de retorno. Analogamente, no servidor é alocada uma porta-servidora para definir o destino do pacote de término. No caso das portas-servidoras, elas podem ser associadas diretamente aos processos fixos, definidos para atendimento concorrente das chamadas remotas.

IV.4.2.4 LIGAÇÃO

Para poder executar uma RPC é preciso ter localizado anteriormente a estação servidora. Nesta implementação a ligação implícita entre clientes e servidores será feita através da interpretação da configuração estática na inicialização do sistema, e poderá ser modificada em tempo de execução através de estensões a serem definidas no futuro. Esta parte será apresentada com detalhe no Capítulo VI, mas é necessário conhecer aqui alguns aspectos que interagem com a execução da RPC.

Será gerado um módulo de configuração para ser carregado em cada estação do sistema distribuido, contendo uma tabela de importação e uma tabela de exportação, que serão utilizadas como interfaces com os "stubs", e um procedimento ligador que representa a interface com o PCRP. Este módulo de configuração é responsável em tempo de execução pela ligação já mencionada.

Na tabela de importação são catalogados todos os módulos importados juntamente com o endereço da estação servidora e um índice que identifica a interface na tabela de exportação do servidor.

Na tabela de exportação deverá existir um apontador, correspondente a cada interface exportada, para um procedimento denominado despachante, implementado no "stub" do servidor e responsável por atender as chamadas endereçadas à interface.

Inicialmente o módulo de configuração chama o PCRP, via InitRCall, passando como parâmetro o nome da estação que é empregado pelo PCRP na ativação da rede, e passa para as estações servidoras, um ponteiro para o procedimento ligador que será chamado pelo PCRP do servidor a fim de encaminhar os pacotes aos "stubs".

No início da execução, os "stubs" do cliente consultam a tabela de importação do módulo de configuração sobre as informações necessárias para a localização da interface importada, que serão guardadas para serem empregadas nas sucessivas chamadas remotas. O "stub" do servidor, na sua fase de inicialização, se responsabiliza por preencher o endereço do despachante na tabela de exportação do módulo de configuração.Ao receber uma RPC, o PCRP da estação servidora encaminhará o pacote de chamada ao módulo de configuração, que identificará e chamará o despachante apropriado.

Como vemos, o módulo de configuração, o PCRP, e os "stubs" têm uma interação grande na execução de uma RPC, e suas interfaces serão apresentadas detalhadamente na descrição do gerador de "stubs" e do interpretador da linguagem de configuração. Esta interação pode ser mais facilmente visualizada através da Figura (IV.21) da seção IV.4.3.5, onde será descrita a execução de RPC.

IV.4.2.5 TRATAMENTO DE EXCEÇÕES

O PCRP sinaliza localmente duas situações de execução no cliente, quando a rede não consegue transmitir o pacote de chamada, e quando expira o tempo de espera para recepção do pacote de retorno.

Outros dois casos são registrados mas não sinalizados, quando há erro na recepção de um pacote, o que causará simplesmente a perda do pacote, e erro na partida da rede que provocará o término do programa.

Outras situações de erro, que mereçam ser notificadas a diferentes estações remotas, devem ser sinalizadas através dos pacotes no campo Codigo-Erro da Figura (IV.15) para serem tratados pelo "stub" do cliente, e serão transparentes para o PCRP, como por exemplo quando o ligador, no servidor, deteta inconsistência na configuração.

IV.4.2.6 ANÁLISE CRÍTICA

DA SILVA analisa em [45] algumas restrições da implementação, em parte ocasionadas pela linguagem Modula-2 e em parte pelo objetivo de obter um resultado de custo mais baixo e mais simples, assim como subsídios para futuras implementações.

A primeira restrição apontada é a do sistema não prever a possibilidade de receber pacotes fora da ordem, já que em ambientes de interrede é comum obter atrasos na transmissão de pacotes que podem seguir rotas diferentes.

Outra restrição que não parece ser muito grave, é o fato de ter limitado o tamanho das mensagens transmitidas a caber em apenas um pacote.

As críticas relacionadas a Modula-2 se referem ao fato de que a linguagem, por estar voltada fundamentalmente a arquiteturas monoprocessadas, não oferece estrutura para suporte á programação distribuida. As primitivas para concorrência são pobres e não existem mecanismos para o tratamento de exceções. Alguns destes problemas foram contornados através das extensões propostas neste trabalho.

Uma sugestão apontada para aumentar a eficiência do sistema é programar o núcleo de multiprogramação como módulo interno ao

PCRP, obtendo maior flexibilidade e integração, e melhorando a implementação do conceito de processo, que está atualmente dividida, tendo as primitivas de criação, destruição e sincronização no núcleo de multiprogramação, e a troca de mensagens através de portas no PCRP.

Ele sugere também a possibilidade de implementar múltiplas interfaces alternativas de diferentes níveis, permitindo que em níveis mais baixos a RPC não seja transparente, e podendo alterar elementos internos do protocolo, como temporização ("timeout"), modo de operação, etc.

IV.4.3 DESCRIÇÃO DO GERADOR DE "STUBS"

IV.4.3.1 INTRODUÇÃO

O gerador de "stubs" tem a finalidade de produzir automaticamente para a implementação de uma RPC dois módulos escritos na linguagem Modula-2, que serão responsáveis para transferir uma chamada de um cliente para um servidor, situados em um nó físico diferente. Estes módulos são denominados "stub" do cliente e "stub" do servidor. Para montá-los, o gerador de "stubs" precisa identificar os módulos exportadores, dos quais as declarações dos procedimentos exportados estão contidas no módulo de definição, ou interface, correspondente.

Quando um módulo de definição é compilado, cria-se um arquivo contendo uma tabela de símbolos com o nome de todos os objetos exportados (constantes, tipos, variáveis e procedimentos). O arquivo de símbolos gerado possui um cabeçalho com a chave do módulo e sua identidade (nome do módulo). Ele é dividido em blocos que correspondem ou à especificação dos objetos dos módulos importados, ou do próprio módulo que está sendo declarado. Cada bloco possui também um cabeçalho com sua identidade.

Foi decidido que o gerador de "stubs" aceitará de entrada os módulos de definição já compilados, ou seja, o arquivo de símbolos, devido ás seguintes vantagens:

i) não há necessidade de verificação da sintaxe da linguagem, pois a mesma já é feita pelo compilador;

ii) todos os objetos estão na tabela de símbolos, que possui entrada dos nomes, sendo fácil a leitura dos mesmos;

iii) a apresentação dos módulos importados fica disponível para possíveis verificações de objetos;

iv) evita-se erros de tipos, pois estes são verificados pelo compilador das interfaces, que são fornecidas a nível da linguagem.

O gerador de "stubs" está dividido em três passos:

i) leitura do arquivo de símbolos para obtenção das informações dos procedimentos exportados e de seus parâmetros;

ii) geração do "stub" do cliente;

iii) geração do "stub" do servidor.

Esta implementação está sendo desenvolvida por LAGES [88] como parte da sua tese de Mestrado.

IV.4.3.2 OBTENÇÃO DOS DADOS UTILIZADOS PELO GERADOR

Durante o primeiro passo, o gerador obtém, do arquivo de entrada, o nome do módulo e o número de sua chave. Estes dados são guardados em variáveis para fornecer os nomes dos "stubs" e para identificar o bloco que corresponde à interface dentro do arquivo. Ao encontrar o bloco, o gerador verifica se existe exportação de variável, e neste caso não será gerado nenhum "stub", já que módulos que compartilham variáveis precisam estar alocados no mesmo nó físico.

Não ocorrendo exportação de variáveis, o gerador procura o símbolo correspondente ao procedimento exportado, preenchendo campos de uma tabela que contém todas as informações para a geração dos "stubs".

IV.4.3.3 GERAÇÃO DO "STUB" DO CLIENTE

O "stub" do cliente é um módulo de implementação que simula

o módulo exportado pelo servidor. Ele precisa conter as entradas correspondentes a todos os procedimentos exportados.

O gerador de "stubs" passa a montar este módulo utilizando o nome, que foi lido no início do arquivo de símbolos, e representa a identidade do módulo. São acrescentadas as importações necessárias para a implementação das chamadas remotas, pois o "stub" do cliente interage com o NUCLEO, o PCRP e com o módulo de configuração.

O gerador acrescenta no "stub" a declaração dos procedimentos que forem exportados. Eles precisam estar declarados da mesma forma que na implementação real, pois representam os pontos de entrada do "stub". O gerador obtém o nome de cada procedimento e os tipos dos parâmetros da tabela montada durante o primeiro passo. Cada procedimento executará as operações necessárias para a realização da chamada remota. Para simplificação do "stub", todas as funções comuns serão concentradas num único procedimento. Estas funções são:

i) alocação de um pacote através do PCRP;

ii) preenchimento dos campos do pacote referentes ao nó físico do "stub" do servidor, índice da tabela de exportação, o número total de bytes que serão transferidos, e o número do procedimento chamado;

iii) preenchimento do campo do pacote referente aos argumentos, depois deles terem sido ordenados;

iv) chamada da rotina do módulo PCRP responsável pela transferência do pacote para o nó físico do servidor;

v) recepção dos resultados com verificação de erro através do NUCLEO;

vi) retirada dos resultados do pacote para as variáveis de retorno, no caso de não ocorrência de erros;

vii) desalocação do pacote através de PCRP.

O corpo principal do "stub", que corresponde à sua inicialização, contém apenas uma consulta à tabela de importação do módulo de configuração, para obtenção de algumas informações

necessárias ao preenchimento do pacote.

IV.4.3.4 GERAÇÃO DO "STUB" DO SERVIDOR

O "stub" do servidor é um módulo, que por não exportar nenhum objeto, não precisa estar separado em duas partes: definição e implementação.

O "stub" será um módulo programa e seu nome será constituido da identidade do módulo interface acrescido da palavra "server". Ele interage com o PCRP, o módulo de configuração e o módulo exportador da interface.

O corpo do "stub" do servidor é constituido por um procedimento despachante que possui um comando CASE cujas alternativas correspondem aos procedimentos exportados, entre as quais será feita uma seleção, baseada em um número transmitido no pacote de chamada.

Cada entrada do CASE realizará as seguintes funções:

i) copiar o campo do pacote, referente aos argumentos, para os parâmetros do procedimento que será chamado;

ii) fazer a chamada local;

iii) receber os resultados e colocá-los no campo do pacote de resposta correspondente.

A inicialização do "stub" do servidor inclui uma chamada ao módulo de configuração para preencher a tabela de exportação deste módulo, informando o endereço do procedimento despachante do "stub" do servidor que receberá o pacote enviado pelo "stub" do cliente.

IV.4.3.5 EXEMPLO ILUSTRATIVO DA IMPLEMENTAÇÃO DE RPC

Vamos considerar um programa escrito em Modula-2, que represente um cliente que deseja importar o procedimento LePag, que faz a leitura de uma página de um determinado arquivo. Este procedimento é exportado por um módulo que contém um servidor de arquivos denominado ServArq, alocado num nó físico diferente. O

programa do cliente, neste caso, não exporta nenhum procedimento, e portanto será um módulo programa. A interface do módulo ServArq é ilustrada de forma resumida pela Figura (IV.16).

```
DEFINITION MODULE ServArq;
EXPORT QUALIFIED LePag, LeCarac,...,PAGINA, BUFFER, ARQUIVO;
  TYPE      PAGINA =
            BUFFER =
            ARQUIVO =
  PROCEDURE LePag(pag:PAGINA;arq:ARQUIVO;VAR buf:BUFFER);
  PROCEDURE LeCarac(...
  .
  .
END ServArq.
```

FIGURA IV.16 - MÓDULO DE DEFINIÇÃO DE ServArq

O módulo de implementação ServArq, que contém o procedimento LePag, é ilustrado de forma resumida pela Figura (IV.17).

```
IMPLEMENTATION MODULE ServArq;
  .
  .
PROCEDURE LePag(pag:PAGINA;arq:ARQUIVO;VAR buf:BUFFER);
  BEGIN
    .
    .              (*Implementação real de LePag*)
    .
  END LePag;
PROCEDURE LeCarac(.....);
  BEGIN
    .
    .
    .
  END LeCarac;
 .
 .
END ServArq.
```

FIGURA IV.17 - MÓDULO DE IMPLEMENTAÇÃO DE ServArq

Não houve preocupação com a definição dos tipos PAGINA, BUFFER e ARQUIVO. O cliente deve importá-los porque são os tipos dos parâmetros do procedimento a ser chamado remotamente. Desta forma, o módulo do cliente terá, de forma resumida, a estrutura apresentada pela Figura (IV.18). Foram utilizados os mesmos nomes para os parâmetros, por uma questão de simplicidade.

```
MODULE Leitor;
FROM ServArq IMPORT LePag,PAGINA,ARQUIVO,BUFFER;
  .
BEGIN
    .
    .
    LePag(pag,arq,buf);
      .
      .
 END Leitor
```

FIGURA IV.18 - MÓDULO DO CLIENTE

A partir da interface do módulo ServArq, o gerador de "stubs" construirá os "stubs" do cliente e do servidor que serão ilustrados pelas Figuras (IV.19 e IV.20) respectivamente.

```
IMPLEMENTATION MODULE ServArq; (*STUB DO CLIENTE*)
  FROM SYSTEM IMPORT BYTE;
  FROM CONFIG IMPORT TipTabImp,TipImp,TabImp;
  FROM NUCLEO IMPORT DWORD, ERROR;
  FROM  PCRP  IMPORT  PACOTE,   RCall,   ObjetoPacote,  AlocaPac,

  PROCEDURE LePag(pag:PAGINA;arq:ARQUIVO;VAR buf:BUFFER);
    BEGIN
      (*ordena parâmetros para variável par*)
      Despachante (par,nroproc);
      (*retorna para buf o resultado*);
    END LePag;

  PROCEDURE LeCarac(...);
    BEGIN
      .
      Despachante (par,nroproc);
      .
    END LeCarac;

  PROCEDURE Despachante (VAR param:ARRAY[0..580] OF BYTE;
            np:CARDINAL);
                        LiberaPac;
   VAR pac : PACOTE;
       nodo,indtab : CARDINAL;
    BEGIN
    AlocaPac(pac);
     WITH pac^ DO
      dest:=nodo;
      mod:=indtab;
      ord:=nroproc;
      arg:=param
    END;
  RCall (pac);
  IF ERROR () THEN ...
  param:=pac.res;
  LiberaPac (pac);
  END Despachante;

BEGIN
    (*Consulta  a tabela TabImp para obter valores das  variáveis
    nodo e indtab*)
END ServArq
```

FIGURA IV.19 - "STUB" DO CLIENTE

```
MODULE ServArqServer; (*STUB DO SERVIDOR*)
  FROM CONFIG IMPORT TipTabExp, TipExp, TabExp;
  FROM PCRP IMPORT PACOTE, ObjetoPacote;
  FROM   ServArq   IMPORT   PAGINA,   ARQUIVO,   BUFFER,   LePag,
                LeCarac,...;

  PROCEDURE Despachante(pac);
   VAR pac:PACOTE;
        PAR0:PAGINA;
        PAR1:ARQUIVO,
        PAR2: BUFFER;
    BEGIN
      CASE pac.ord OF
        1: (*desordena os parâmetros do campo pac.arg)
           (para as variáveis PAR0,PAR1*)
           LePag(PAR0,PAR1,PAR2);
           pac.res:=PAR2;

        2: :
           LeCarac(...);
           :
      END; (*CASE*)
  END  Despachante.

BEGIN
    (*O endereço do procedimento Despachante)
    (é colocado na tabela TabExp*)
END ServArq Server.
```

FIGURA IV.20 - "STUB" DO SERVIDOR

Vamos agora descrever a execução de uma RPC, na nossa implementação, mostrando a interação entre as diferentes interfaces já citadas, através do exemplo da chamada do procedimento LePag pelo módulo Leitor. As setas numeradas da Figura (IV.21) ilustram a sequência de passos que compõem a execução e que listaremos a seguir.

A lista de passos é a seguinte:

1. transferência da chamada para o "stub" do cliente.
2. chamada ao PCRP do cliente passando o pacote com os argumentos do procedimento.
3. passagem do pacote do PCRP do cliente para o PCRP do servidor.
4. chamada do procedimento ligador do módulo de configuração do servidor, passando o pacote.
5. ativação do despachante correspondente à chamada remota passando o pacote contendo os argumentos do procedimento.
6. execução real do procedimento ativado pelo "stub" do servidor (chamada local).

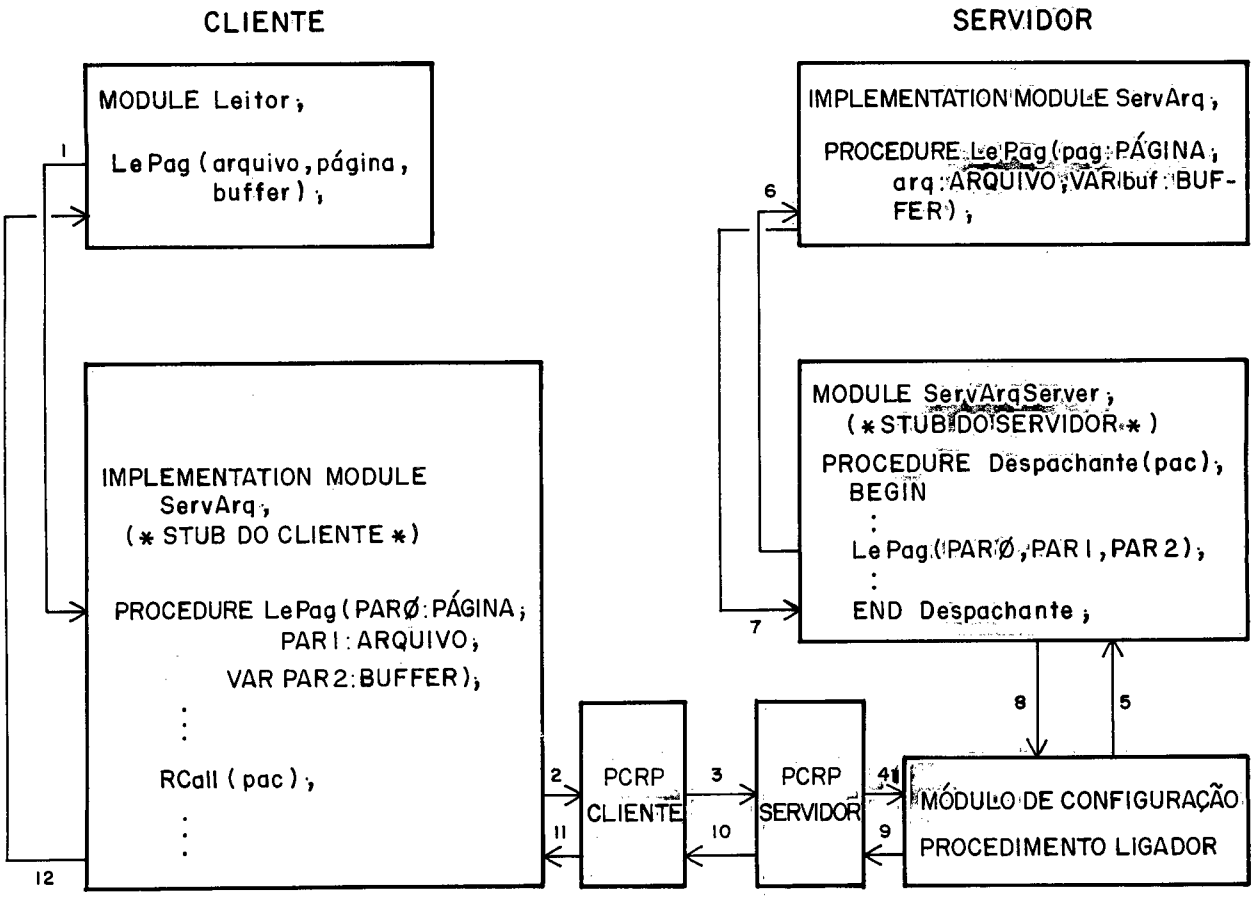

FIGURA IV.21 - ESQUEMA DE EXECUÇÃO DE RPC

7. retorno do controle de execução para o "stub" do servidor com as variáveis atualizadas (resultados).
8. retorno do controle de execução para o procedimento ligador do módulo de configuração devolvendo o pacote com os resultados.
9. retorno do controle de execução para o PCRP do servidor devolvendo o pacote com os resultados.
10. passagem do pacote contendo os resultados para o PCRP do cliente.
11. retorno do controle de execução para o "stub" do cliente devolvendo o pacote com os resultados.
12. retorno do controle de execução para o cliente devolvendo os resultados da chamada remota.

CAPÍTULO V

CONCEITOS GERAIS SOBRE CONFIGURAÇÃO

V.1 INTRODUÇÃO

É amplamente conhecida a deficiência da tecnologia de desenvolvimento de software comparado com o grande avanço da tecnologia de hardware. Isto se reflete no baixo índice de aumento de produtividade do processo de criação de software e na diminuição enorme no preço dos componentes de hardware.

Portanto é de fundamental importância desenvolver ferramentas, que auxiliem a produção de sistemas de software grandes e de vida longa, a partir da fase de projeto do sistema e durante seu ciclo de vida, que inclui a integração, a evolução e a manutenção.

Nos capítulos II e III se demonstrou a importância da estruturação modular dos sistemas, obtida através da combinação de módulos desenvolvidos separadamente, e podendo ser reusados para compor sistemas diferentes e facilmente modificáveis.

Existem várias linguagens, ferramentas de desenvolvimento de software e sistemas operacionais, que dão suporte a algum tipo de interconexão modular. Neste capítulo nos concentraremos nos conceitos principais ligados aos problemas de interconexão modular, ou configuração (como também são denominados na literatura), e mais especificamente nas linguagens que dão suporte a estes conceitos. Foram definidas linguagens especialmente projetadas para dar suporte à interconexão modular denominadas MILs ("Module Interconnection Languages").

Iniciaremos nosso estudo pela análise das funções das MILs, passando depois ao caso específico das características de linguagens de configuração que, como será mostrado, pertencem à família das MILs, salvo algumas diferenças. Depois de analisar a especificação da configuração de um sistema, trataremos das características de configuração estática e dinâmica, abordando com mais detalhe alguns problemas específicos deste último ponto.

Gostaríamos de definir aqui o contexto no qual está baseado nosso estudo. Como foi dito na introdução, nosso objetivo é a

construção de um ambiente de programação distribuida. Dependendo da arquitetura física utilizada, o ambiente poderá ser estruturado de diferentes maneiras. Se considerarmos os computadores utilizados num sistema de controle, geralmente estes serão comparativamente simples e muitas vezes sem memória auxiliar. Neste caso será impossível usar estes computadores para o desenvolvimento de software, e portanto será necessário usar um enfoque do tipo máquina hospedeira/máquinas alvo. O software será desenvolvido numa máquina hospedeira com um sistema operacional de propósito geral que dará suporte a: um sistema hierárquico de arquivos, bancos de dados, editores de telas, pacotes gráficos, impressoras e outras ferramentas associadas com o ambiente de suporte de programação. Depois de confeccionados os programas na máquina hospedeira eles serão carregados nas respectivas máquinas alvo. Muitas vezes é útil dar suporte a uma comunicação contínua entre a máquina hospedeira, e as máquinas alvo, de maneira a poder ainda utilizar de forma remota algumas das facilidades oferecidas pela máquina hospedeira, como, por exemplo, o sistema de arquivos, dispositivos gráficos, serviços de impressão e mecanismos eficientes para depuração. A configuração de hardware de um determinado sistema poderá então estar composta de uma ou mais máquinas hospedeiras e várias máquinas alvo. O sistema de desenvolvimento hospedeiro estará ligado através de uma rede ao sistema alvo, permitindo que as mudanças do sistema possam ser validadas e controladas de forma centralizada.

Por outro lado, se a arquitetura física do sistema estiver composta de máquinas mais poderosas do que as citadas anteriormente, o ambiente de desenvolvimento não precisará estar tão centralizado podendo ter funções específicas em algumas máquinas, compartilhadas pelas outras. Alternativamente podemos ter um conjunto mínimo de funções em cada máquina, de maneira a dar mais autonomia a cada uma. Não se deve esquecer que será sempre necessário algum tipo de controle global do sistema. O tipo de arquitetura física escolhida terá grande influência nos conceitos discutidos neste capítulo, particularmente em relação à configuração dinâmica.

Este estudo deu subsídios fundamentais para a proposta e implementação da linguagem de configuração, definida para construir o ambiente de programação distribuida baseado em

Modula-2, e que será apresentada no próximo capítulo.

V.2 - LINGUAGENS DE INTERCONEXÃO MODULAR

De acordo com o que foi apresentado por PRIETO e NEIGHBORS em [128], o conceito fundamental das linguagens de interconexão modular (MILs) está baseado na diferença entre a programação em larga escala e a programação em pequena escala. Como já foi mencionado no Capítulo III, a diferença principal entre as duas atividades é que "estruturar um conjunto grande de módulos para formar um sistema é uma atividade intelectual essencialmente diferente da de construir os módulos individualmente", (DE REMER e KRON [47]).

A maioria das linguagens amplamente utilizadas (Algol, Fortran, Pascal, Cobol, etc) foram projetadas para programação em pequena escala (LPP); as suas notações são apropriadas para expressar como uma parte (ou módulo) de um sistema realiza a sua função.

A programação em larga escala se preocupa com a forma de construir sistemas a partir de partes já programadas. A linguagem que trata das interligações do fluxo de controle e dos dados entre um conjunto de módulos é denominada de linguagem de programação em larga escala (LPL). Uma MIL é uma LPL com uma sintaxe formal, possível de ser interpretada mecanicamente, que provê ao projetista meios de representar a estrutura completa de um sistema grande, de forma concisa, precisa e verificável.

Segundo foi definido em [128] a especificação completa de um sistema deve incluir três partes:

i) A descrição de cada módulo do sistema (LPP);

ii) A descrição dos recursos providos e requeridos por cada módulo do sistema (MIL, linguagem de recursos);

iii) A descrição do fluxo de recursos entre os módulos do sistema (MIL, linguagem de interconexão).

Na descrição da MIL, os recursos são considerados objetos de intercâmbio entre módulos. Os recursos são entidades que podem ser nomeadas na LPP (variáveis, constantes, procedimentos, definições de tipo, etc) e que poderão ser referenciadas por

outro módulo dentro de um determinado sistema. A segunda parte que acabamos de mencionar está definida pela linguagem MIL, que especifica as interfaces entre os módulos. Comparando isto com as linguagens de LPP já vistas, que permitem definir um módulo separando a parte de interface da parte de implementação, no esquema apresentado pela MIL, a definição da interface faria parte da MIL.

Na maioria dos esquemas de interconexão modular examinados pelos autores citados, a informação correspondente à programação em larga escala está na forma de uma MIL, e a informação correspondente à programação em pequena escala está na forma de uma linguagem convencional de programação (LPP). Entretanto a forma de empacotar esta informação difere nos diversos esquemas. Por um lado existem sistemas definidos por um conjunto de módulos, cada um contendo informação de programação em pequena escala, e da MIL, sem conter uma descrição central do sistema além da lista dos módulos que o compõem. Por outro lado, existem sistemas definidos por um conjunto de módulos, que só contêm informação de pequena escala, e que possuem uma descrição central do sistema contendo toda a informação da MIL para cada módulo e das interconexões do sistema. Em ambos os casos faz sentido compilar a definição da MIL do sistema para verificar se as interfaces dos seus componentes são compatíveis. Nenhuma informação da programação em pequena escala é necessária para realizar essa compilação.

São definidas em [128] as seguintes funções básicas de uma MIL:

i) **Descrição da estrutura do sistema** definindo o escopo dos nomes nos módulos e subsistemas, e especificando a interconexão entre os módulos. Isto é realizado escrevendo a parte de descrição de cada módulo e compilando todas as descrições juntas;

ii) **Definição das conexões estáticas dentro de cada módulo** e teste estático de tipo dentro dos limites do módulo. Estático aqui se refere a tempo de compilação, enquanto dinâmico se refere a tempo de execução;

iii) **Prover diferentes tipos de acesso** aos recursos dos módulos (somente leitura, leitura e escrita, etc) e permitir

que os módulos e/ou os subsistemas estejam escritos em diferentes linguagens de programação, ou que consistam unicamente de texto;

iv) **Gerenciamento de controle de versões** e famílias do sistema.

Além das funções listadas, uma MIL usualmente serve como **ferramenta de gerenciamento de projeto**, estimulando o projetista a estruturá-lo antes de programar os detalhes. Ela serve também como **ferramenta de suporte** no processo de desenvolvimento, permitindo concentrar toda a estrutura do programa e favorecendo a verificação da integridade do sistema antes de iniciar a implementação. A MIL pode prover também uma forma de padronizar a comunicação entre os componentes do sistema e a documentação da estrutura do sistema. Podemos concluir que, do ponto de vista de Engenharia de Software, a MIL representa uma ferramenta poderosa para o processo de desenvolvimento de software.

PRIETO e NEIGHBORS apontam que DE REMER e KRON [47] e THOMAS [167] definiram de forma precisa as funções que são consideradas como não pertencentes ao domínio das MILs, com o objetivo de dintinguir claramente entre uma MIL e outras ferramentas ou linguagens que realizem funções parecidas relacionadas à interconexão de módulos. As funções que não correspondem a uma MIL são:

i) **Carregamento**: a MIL deveria deixar esta função a uma linguagem de carregamento, ou a outras ferramentas do ambiente de desenvolvimento de software;

ii) **Especificação funcional**: a MIL mostra somente a estrutura estática do software e não deveria especificar a natureza de seus recursos. Esta função deveria ser atribuida a outros subsistemas;

iii) **Especificação de tipo**: a MIL se preocupa em mostrar e verificar os diferentes caminhos de comunicação entre módulos de um sistema por meio dos nomes dos recursos. Alguns destes recursos podem ser tipos, mas a MIL lida com os seus nomes e não com as suas especificações;

iv) **Instruções embutidas de ligação-edição**: estas operações

deveriam pertencer ao sistema operacional ou a uma linguagem de comandos separada.

Alguns sistemas integrados de desenvolvimento de software, como por exemplo MESA (MITCHELL [113]) realizam, algumas das funções de interconexão de módulos, assim como algumas das listadas como não pertencentes ao domínio da MIL. Por esta razão tais sistemas não são considerados estritamente como MIL.

A tendência atual no desenvolvimento das MILs é manter o domínio da MIL bem definido, de maneira que MILs independentes possam ser desenvolvidas e logo integradas como parte de um ambiente de desenvolvimento de software, como por exemplo GANDALF (HABERMAN et alii [63]). O sistema composto por MESA e C/MESA também corresponde a esta tendência, com a ressalva que o sistema está restringido ao uso de uma única linguagem de programação para a codificação dos módulos.

A discussão destes conceitos de MIL é da maior relevância para analisar as características necessárias para definir a configuração de um sistema distribuido, e em particular nos ajuda na definição da linguagem de configuração (LPL), para a construção do ambiente de programação distribuida baseado em Modula-2.

V.3 MODELO DE CONFIGURAÇÃO

V.3.1 ESPECIFICAÇÃO DA CONFIGURAÇÃO

Como já foi mencionado anteriormente, consideramos sistemas que são construidos a partir de um conjunto de componentes ou módulos programados de forma separada e independente. Assim como o texto de um programa fonte escrito numa linguagem de programação em pequena escala descreve o comportamento e a estrutura de um módulo de software, a especificação da configuração escrita numa linguagem de configuração descreve o comportamento e a estrutura do sistema composto pelo conjunto de componentes. A linguagem de configuração pertence então à família das MILs, discutidas na seção anterior, mas apresentaremos aqui algumas características específicas.

Para o caso de sistemas distribuidos, que é de nosso

interesse neste trabalho, a especificação completa da configuração de um sistema composto de um conjunto de componentes executando em vários nós físicos separados, cooperando para implementar alguma aplicação, consistirá da descrição do número e tipos de nós disponíveis, do número e tipos dos componentes executando nesses nós, dos modelos de comunicação entre os componentes, e da relação entre os componentes e os nós, isto é que componentes estão carregados em cada nó físico. Para sistemas não distribuídos, logicamente será omitida a descrição dos nós e o mapeamento dos componentes nos nós.

Como foi apresentado por MAGEE em [104] podemos dividir a especificação da configuração de um sistema distribuído em três seções:

i) **estrutura lógica**: descreve os tipos dos componentes de software, a partir dos quais o sistema é construído, as instâncias destes componentes que existem no sistema, e como essas instâncias estão interligadas, ou seja, qual instância se comunica com quais outras. Pode fazer parte também da especificação de cada componente a informação sobre os requisitos de recursos, tais como memória, tipo de processador e periféricos.

ii) **estrutura física**: descreve os componentes de hardware e a estrutura da sua interconexão física. Os componentes físicos são os que processam os componentes lógicos. Associada a cada componente de hardware pode estar a especificação dos recursos por ele providos, tais como processadores, memória e periféricos.

iii) **mapeamento entre a estrutura lógica e a física**: descreve a locação física dos componentes lógicos. Para obter flexibilidade deve ser possível alocar um ou mais componentes lógicos num componente físico. A única restrição no mapeamento deve estar relacionada aos requisitos dos recursos dos componentes lógicos alocados num componente físico que não podem exceder os recursos por ele providos.

Neste trabalho concentraremo-nos na parte da especificação da estrutura lógica e do mapeamento entre esta estrutura lógica

e a física

V.3.2 CONFIGURAÇÃO ESTÁTICA

Na maioria dos sistemas implementados em software, tanto nos centralizados quanto nos distribuídos, a configuração é estática: isto é consequência da maneira pela qual os programas são construídos; eles não podem ser modificados sem reconstruir o sistema. Retomando a analogia com a programação em pequena escala, da mesma forma que os componentes de software são compilados a partir de seus programas em código fonte, os sistemas distribuídos podem ser construídos a partir de suas especificações de configuração.

A Figura (V.1) de [104, 87] ilustra a forma de configuração estática, na qual é construído um sistema a partir da sua especificação de configuração. Nesta figura diferenciamos os recursos, representados por retângulos, dos processos representados pelas formas ovais.

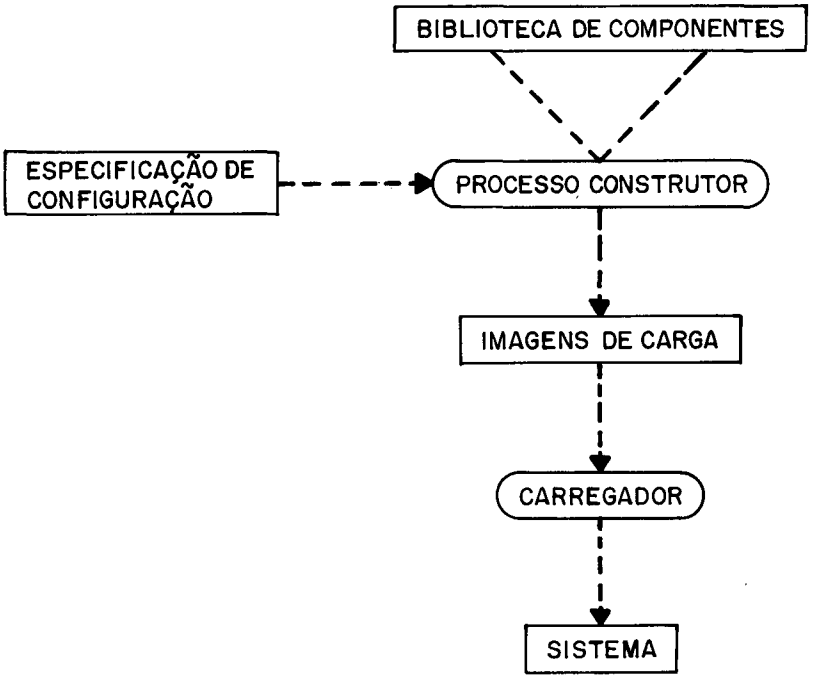

FIGURA V.1 - PROCESSO DE CONFIGURAÇÃO ESTÁTICA

O processo construtor é dirigido pela especificação da configuração, para produzir as imagens de carga dos grupos de componentes para cada estação física do sistema distribuido. Este processo construtor é equivalente ao ligador dos sistemas de programação sequenciais tradicionais.

Este enfoque de construção de sistemas provê uma configuração estática completa. É completa no sentido que todos os componentes do sistema são configurados ao mesmo tempo. Se for necessário fazer alguma modificação no sistema, ele precisa ser parado completamente e ser reconstruido de acordo com uma nova especificação de configuração. Esta nova especificação de configuração será uma nova versão da configuração anterior incorporando as mudanças realizadas. Podemos notar que este enfoque de construção de sistema é equivalente ao processo de edição, compilação, ligação e carregamento utilizado para a programação sequencial tradicional. Ele não suporta mudanças que possam ser realizadas sem parar o sistema como um todo, caso este que se apresenta na prática em sistemas de controle e sistemas de tempo real.

Analisaremos nas próximas seções os conceitos de configuração dinâmica e os problemas envolvidos na sua implementação, sendo que vários deles estão ainda em aberto.

Queremos salientar que na literatura encontramos poucas linguagens que dão suporte à configuração dinâmica, e que a maioria implementa configuração estática. Entre as linguagens mais conhecidas que implementam configuração estática, e que serão tratadas com mais detalhe em seções posteriores, podemos citar ADA (ICHBIAH [73]), DP (BRINCH HANSEN [29]), SR (ANDREWS [4]), *MOD (COOK [42]), CSP (HOARE [68]), entre outras. Em geral estas linguagens estão restritas à configuração estática porque não permitem uma separação nítida entre a programação dos componentes e a configuração. Por exemplo, nas linguagens mencionadas, na comunicação entre processos é preciso especificar diretamente o correspondente. As relações de interconexão estão então embutidas no nível de programação e portanto não facilitam as mudanças de configuração. Este tema será discutido com mais detalhe, já que não existe ainda consenso total, sobre a necessidade de separação completa das duas linguagens (LPP e LPL).

Como foi citado por KRAMER e MAGEE em [87], algumas linguagens oferecem interconexões dinâmicas através da passagem de nomes. Por exemplo, em SR uma mensagem pode conter o nome de uma entrada de processo para a qual mensagens posteriores podem ser enviadas. Mas não é possível prover interconexões imprevisíveis sem reprogramar, recompilar e reconstruir o sistema. Veremos no Capítulo VII como na nova versão de SR (ANDREWS e OLSSOM [6]) foram feitas modificações para que a linguagem dê suporte à configuração dinâmica. O uso do termo estático é mais difícil de justificar em linguagens que permitem criação dinâmica de componentes de software, como em ADA. Entretanto, em ADA os componentes que podem ser criados dinamicamente são de tipos já existentes. Modificações ou extensões do sistema através da modificação de componentes já existentes, ou através da introdução de novos tipos de componentes, também ocasionarão a reprogramação e reconstrução do sistema. No máximo, somente aquelas estações que não são afetadas pela nova especificação de configuração não precisarão ser paradas e recarregadas.

Podemos citar também modelos que possuem linguagens de configuração separadas, podendo ser utilizados para implementar configuração estática, tais como CONIC/C (KRAMER e MAGEE [87], DULAY et alii [52]), Mascot3 (BATE [16]), C/MESA (MITCHELL et alii [113]) e a proposta de adaptação de Modula-2 para sistemas distribuídos de MELLOR, DUBERY e WHITEHEAD [111]. Algumas destas linguagens mencionadas serão analisadas detalhadamente no Capítulo VII.

Podemos destacar que existem alguns sistemas operacionais distribuídos que suportam configuração dinâmica, tais como UNIX [80], Roscoe [153], CHORUS [15], Medusa [122] e StarOs [75]. No entanto, se isto não está integrado com um sistema que contenha uma linguagem de configuração e de programação, não é possível validar as configurações testando a compatibilidade das interfaces dos componentes.

V.3.3 CONFIGURAÇÃO DINÂMICA

Analisamos na seção anterior os conceitos de configuração estática. Entretanto, é necessário notar que sistemas

distribuidos grandes e de vida longa útil dificilmente permanecerão estáticos. Estes sistemas estão sujeitos a mudanças que, como foi colocado em [104] por MAGEE, podem ser caracterizadas em duas classes, como já foi mencionado no Capítulo II.

As **mudanças denominadas de evolucionárias** são as devidas a novas necessidades humanas, a mudanças no ambiente da aplicação e ao surgimento de novas tecnologias a serem incorporadas. De fato, a introdução de novos elementos mais avançados de hardware estimulam mudanças no ambiente da aplicação e dos serviços oferecidos pelo sistema, precisando modificar funções já providas pelo sistema ou estendendo-o com novas funções. Podem servir também para implementar técnicas aperfeiçoadas e prover redundância no sistema.

Por outro lado, podemos chamar de **mudanças operacionais** as provocadas por falhas de partes do sistema, o qual precisará ser reestruturado para isolar essas falhas, ou por redimensionamento do sistema que causará uma nova distribuição das partes do sistema e possivelmente também incorporação ou remoção de componentes. A diferença principal entre estas duas classes de mudanças é que as mudanças evolucionárias não são previsíveis na hora de projetar o sistema, e portanto não podem ser tratadas sem adicionar novas funções ao sistema.

Portanto, deve ser possível construir um sistema distribuido que possa ser reconfigurado acrescentando, substituindo, retirando ou trasladando componentes, sem ter que deter a execução do sistema como um todo para reconstruí-lo. A configuração dinâmica é a habilidade de criar ou destruir componentes, e de controlar as suas interconexões , isto é, de modificar e estender o sistema, enquanto o sistema como um todo continua executando e prestando seus serviços. Em alguns casos não é possível, seja por razões econômicas ou de segurança, parar o sistema inteiro para modificar uma parte dele.

Para satisfazer estas características de configuração dinâmica, é necessário encarar os seguintes problemas relacionados com a configuração e a operação do sistema, como foi levantado por DAY em [46], além dos já analisados na configuração estática:

i) **Expressar as alterações da configuração do sistema:** expressar como o novo sistema difere do anterior que já está executando.

ii) **Realizar a reconfiguração dinâmica do sistema:** a partir do sistema que já está funcionando, da especificação das mudanças e da disponibilidade dos seus componentes, realizar as mudanças indicadas; garantir que a disponibilidade e a conflabilidade do sistema não sejam degradadas pela reconfiguração dinâmica.

iii) **Prover suporte de linguagem para a reconfiguração dinâmica:** fornecer ferramentas e poder expressivo à linguagem utilizada para programação distribuída para implementar a reconfiguração dinâmica.

Em [104, 87] é apresentado um modelo para o processo de configuração dinâmica que trata de modificações e extensões não previsíveis, sem precisar reconstruir o sistema como um todo. Este modelo supõe, que sempre que seja possível, as mudanças sejam realizadas sem parar as partes não afetadas do sistema, e o seu processo é ilustrado pela Figura (V.2), diferenciando processos e recursos de forma análoga à Figura (V.1).

A especificação da configuração descreve a estrutura lógica e física do sistema. As mudanças do sistema são submetidas na forma de especificação de mudanças, tais como a introdução de componentes novos, modificações de componentes já existentes, e introdução de modelos de interconexão diferentes.

As mudanças precisam ser validadas para ter certeza que elas estejam compatíveis com o sistema existente. O resultado é uma nova especificação. O gerenciador de configuração gera os comandos necessários para o sistema operacional carregar o código apropriado, tirar e criar componentes, e atualizar as conexões, isto é, desligar e ligar conexões. O resultado é uma descrição de sistema que corresponde à nova especificação. Este processo de configuração dinâmica deve ter o suporte da linguagem utilizada para programar os componentes do sistema, e do sistema

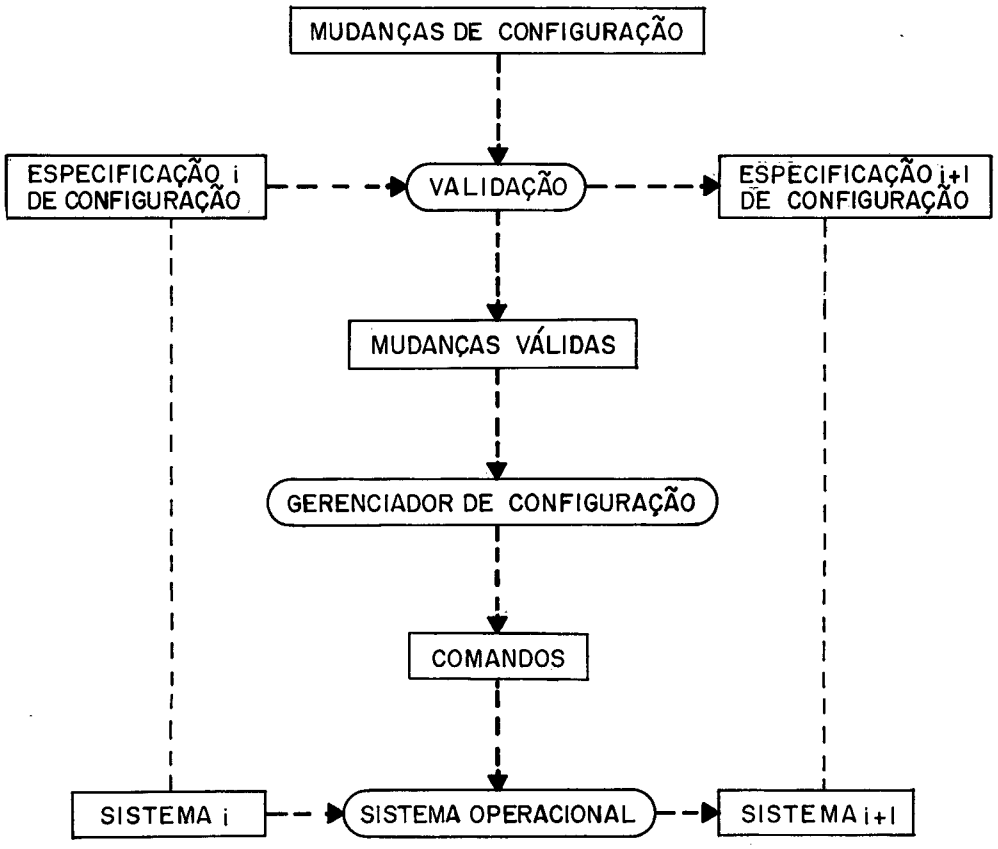

FIGURA V.2 - PROCESSO DE CONFIGURAÇÃO DINÂMICA

operacional subjacente. O sistema operacional deverá fornecer, por exemplo, primitivas para ligar e desligar ligações, para tirar, carregar e inicializar componentes, e para a implementação da comunicação entre elas. Este modelo de configuração dinâmica satisfaz os requisitos de flexibilidade já identificados no Capítulo II: a flexibilidade funcional, a topológica, a de implementação e a de tempo.

Exemplos de linguagens de programação existentes para um único processador, que provêem uma linguagem separada de configuração e algum grau de configuração dinâmica são MESA e C/MESA (MITCHELL et alii [113]) e Mascot (SIMPSON e JACKSON [150]). Na área de sistemas distribuídos, CONIC (KRAMER et alii [86]) também fornece uma linguagem separada de configuração

CONIC/C, e uma forma de configuração dinâmica, que tem evoluido nos últimos anos e será tratada com mais detalhe numa próxima seção e no Capítulo VII. Podemos citar também a linguagem Mascot3 (BATE [16]), que foi estendida, a partir de Mascot, para sistemas distribuidos. No trabalho citado é mencionada a possibilidade de seu uso para configuração dinâmica, apesar de não ter sido apresentada a forma de fazê-lo, nem a sua implementação. Nesta área de sistemas distribuidos, podemos citar três linguagens que fornecem configuração dinâmica, ARGUS (LISKOV e SCHEIFLER [97]), NIL (BURGER [33], PARR e STROM [126]) e SR (ANDREWS e OLSSOM [6]), que foi estendida a partir da versão anterior, sendo que nas três os comandos de configuração estão embutidos no texto dos componentes dos programas. Algumas destas linguagens serão analisadas detalhadamente no Capítulo VII. O projeto Cnet (FANTECHI et alii [54], INVERARDI et alii [74]), como já foi mencionado no Capítulo IV, também acrescentou à linguagem ADA mecanismos de comunicação e sincronização e facilidades para configuração dinâmica, através do uso de pacotes ("packages") específicos para implementar estas funções e facilitar o uso de ADA em sistemas distribuidos.

V.4 PROPRIEDADES NECESSÁRIAS PARA CONFIGURAÇÃO

Pretendemos nesta seção fazer um levantamento das propriedades do modelo de configuração, para realizar as configurações estática e dinâmica já descritas nas seções anteriores. Queremos salientar que nos concentraremos particularmente nas linguagens de programação e que algumas destas propriedades já foram analisadas no Capítulo II, mas estudaremos aqui com maior detalhe as mais relacionadas aos conceitos de configuração. Esta análise está baseada no estudo realizado por KRAMER e MAGEE [104, 87].

V.4.1 LINGUAGEM DE PROGRAMAÇÃO

Já foram analisadas algumas das propriedades necessárias para programação distribuida, das linguagens utilizadas para a programação dos componentes, a partir dos quais será construido o sistema.

Vimos a importância da linguagem prover a programação de unidades de software, habitualmente chamadas **módulos**, que possam ser escritos e compilados separadamente, fornecendo **independência de contexto**, já que não dependerão da configuração na qual o sistema será executado. Os comandos dentro de um módulo devem referenciar unicamente objetos locais, já que referências a objetos globais dificultam a possibilidade de acrescentar ou remover módulos, e restringe, no caso de sistemas distribuídos, as formas de alocar módulos a recursos físicos.

Outra característica importante é a **independência de interconexão**. Para satisfazer esta propriedade, é preciso que não haja nomeação direta de outros módulos ou entidades de comunicação, porque no caso de acontecerem mudanças, a nomeação direta implicaria em mudanças no texto dos programas e recompilação das partes modificadas, restringindo a flexibilidade lógica do sistema. É então necessário que um módulo só se comunique com o mundo externo através da sua interface, e que não contenha informação sobre as interconexões, embutida no texto do módulo. A propriedade de independência de interconexão é a propriedade chave para separar a programação dos módulos da configuração, como foi colocado por DULAY em [56].

Analisamos anteriormente a importância dos módulos possuirem uma **interface bem definida**, que vem a ser a única informação lógica requerida ao nível da configuração para construir o sistema a partir dos módulos já programados. Desta forma, a interconexão entre módulos é especificada e testada ao nível da configuração usando a informação das interfaces, isto é, o tipo da informação e o mecanismo pelo qual ela é transferida.

A fim de obter uma flexibilidade de configuração total, é fundamental a propriedade de **transparência de comunicação**. Para isto, é necessário que as primitivas de comunicação entre módulos, fornecidas pela linguagem de programação (LPP), tenham a mesma sintaxe e semântica para comunicação local (na mesma estação física) que para comunicação remota (entre estações físicas diferentes). Esta transparência total nem sempre é conseguida, como já foi analisada anteriormente, e queremos salientar que estamos nos referindo ao comportamento lógico, já que é inevitável a diferença de desempenho da comunicação remota.

Outra propriedade desejável é poder especificar os

requisitos de recursos do módulo, para que estes sejam conhecidos na fase de compilação. Isto permite ao gerenciador de configuração testar se a estação, na qual o módulo será carregado, possui os recursos necessários. No caso de aplicações em tempo real esta propriedade é essencial para que o sistema tenha um comportamento determinístico confiável.

Este conjunto de propriedades define o grau de **independência de configuração** dos componentes escritos na linguagem de programação. Várias linguagens de programação satisfazem algumas mas não todas estas propriedades. Por exemplo, linguagens que fornecem o conceito de módulo ou outro parecido, como SR (recursos), Modula-2 (módulos), ARGUS (guardiães), CONIC (módulos), MESA (módulos), apresentam a independência de contexto. Entretanto, quando algumas destas linguagens, tais como ADA, MESA e Modula-2 declaram, como parte do contexto, módulos contendo dados e procedimentos que serão compartilhados por vários outros, esta independência não é mais totalmente satisfeita. Os módulos de CONIC apresentam independência de interconexão através do uso de portas locais de entrada e saída onde as mensagens são enviadas e recebidas. Linguagens tais como ADA, *MOD, CSP e SR, que usam nomeação direta não apresentam esta propriedade. A maioria das linguagens de programação modernas que fornecem estruturas de módulos, provêem interfaces bem definidas usando um mecanismo similar ao de importação e exportação de listas de identificadores de MESA, como por exemplo CONIC, Modula-2, SR.

V.4.2 LINGUAGEM DE CONFIGURAÇÃO E ESPECIFICAÇÃO DAS MUDANÇAS

Apresentaremos agora as propriedades necessárias para que uma linguagem de configuração estática descreva a estrutura lógica de um sistema, e as propriedades para especificar as mudanças do sistema a fim de realizar a configuração dinâmica.

A linguagem de configuração deve especificar o conjunto dos tipos de módulos e os tipos das sub-configurações, a partir do qual será construido o sistema, isto é, definir o contexto.

Deve especificar as instâncias dos módulos e das sub-configurações que são criadas no sistema, indicando a sua locação lógica, que deverá depois ser mapeada na estrutura física na qual

ele será executado.

Deve também descrever as interconexões entre as instâncias de módulos e de sub-configurações que formam o sistema.

Para que a linguagem de configuração possa expressar as mudanças da configuração, é necessário que ela possa especificar as funções inversas às que acabamos de citar. Para isto ela deve poder desligar as instâncias de módulos e sub-configurações, destruir essas instâncias e remover os tipos de módulos e sub-configurações do contexto do sistema. As mudanças devem se apresentar como atualizações das especificações da configuração.

Para obter uma flexibilidade de configuração total é necessário que cada uma das funções de configuração listadas seja especificada separadamente. Se por exemplo é combinada a instanciação de um módulo com as suas interconexões, não será possível modificar as interconexões sem destruir a sua instância e depois reinstanciá-la com as novas interconexões.

Na descrição das funções, temos mencionado a existência de sub-configurações além dos módulos, como componentes a partir dos quais o sistema será construído. Num sistema grande é fundamental a necessidade de estruturação para facilitar o processo de especificação e de compreensão das especificações.

Outra característica importante a ser definida na linguagem de configuração ,tanto estática quanto dinâmica, é a **unidade de configuração**, que por sua vez será a unidade de substituição para configuração dinâmica. Normalmente são definidas como unidades de configuração e substituição as unidades básicas de estruturação das linguagens, por serem as mais facilmente reconhecidas e por facilitar a verificação de tipos. Mais recentemente, ligado ao problema de estruturação de sistemas, foi levantada a necessidade de definir unidades mais estruturadas, como é colocado por BLOOM em [23] para a linguagem ARGUS, e por MAGEE et alii em [106] para o sistema CONIC. Dada a importância deste ponto, o discutiremos com mais detalhe numa próxima seção.

Uma propriedade desejável é que a especificação da configuração seja descritiva, isto é, declarativa, e não operacional, no sentido de ser um programa de configuração que seja executável. Uma linguagem operacional para especificação de um sistema no alto nível supõe uma máquina abstrata que executa a especificação. A semântica dos comandos da linguagem é dada pela

transição dos estados da máquina, sendo que cada estado representa a configuração num determinado tempo. As funções de transição dão o significado dos comandos da linguagem, como é colocado por COHEN et alii em [41]. Em geral, as especificações declarativas são mais fáceis de analisar e manipular, por exemplo, para verificação de equivalências, verificação de consistência lógica e transformações. A especificação precisa descrever a configuração corrente do sistema. As especificações operacionais introduzem uma dependência com o tempo que dificulta a verificação da consistência do sistema. Adicionalmente, a introdução de mudanças arbitrárias imprevisíveis podem ser representadas por especificações de atualizações declarativas. Um programa de configuração executável, que formasse parte do comportamento do sistema, apresentaria dificuldades se fossem permitidas mudanças em tempo de execução.

Gostaríamos de mencionar, sem entrar em detalhes, que as propriedades apresentadas, tanto para a linguagem de programação (LPP) quanto para a de configuração (LPL), precisam ter o suporte de um sistema operacional, como já foi mencionado antes, que permita fazer o gerenciamento dos módulos, o gerenciamento das conexões e a implementação da comunicação entre os módulos. É necessário também que o sistema operacional facilite as modificações em tempo real, preveja comunicação direta entre cada par de estações físicas e ofereça flexibilidade de configurações. Como já foi citado anteriormente, as configurações estática e dinâmica podem ser tratadas segundo dois modelos diferentes: de forma centralizada ou distribuída. Neste último caso as funções básicas do sistema operacional deverão ser oferecidas por um núcleo mínimo carregado em cada estação física.

Todos os modelos, que provêem uma linguagem separada de configuração, dão suporte à definição de contexto, à instanciação e à interconexão. Entretanto C/MESA, a linguagem de configuração de MESA, combina instanciação com interconexão. Em C/MESA, as instâncias dos módulos são interligadas, parametrizando-as com os arquivos de interfaces, cujas definições são importadas ou exportadas pelos módulos do programa. Um método parecido de interconexão é utilizado em Mascot, no qual as atividades (processos) são parametrizados com as áreas de dados de

intercomunicação (IDA). Portanto, tanto MESA quanto Mascot não apresentam a propriedade de separação de funções. Em CONIC esta separação existe já que as portas das instâncias de módulos são interconectadas independentemente através da declaração **LINK** da linguagem de configuração.

V.4.3 PROCESSO DE VALIDAÇÃO

Os testes que podem ser feitos, para validar tanto a configuração quanto as mudanças de configuração, dependem principalmente das interfaces providas pelos módulos de software. A verificação mínima, que deve ter o suporte de ambas as linguagens de programação e configuração, consiste na capacidade de testar a compatibilidade de tipo entre os módulos que se comunicam.

É fundamental garantir que a configuração do sistema presente seja dada e satisfaça a especificação corrente. O sistema pode ser considerado como uma implementação da especificação, e pode ser verificado na forma convencional de programas. Em [104, 87] é mostrado como as mudanças de especificação podem ser verificadas, usando o método de verificação descrito por HOARE para representação de dados em [66]. Uma função abstrata A pode ser definida para mapear o sistema com a sua especificação:

A(sistema i) = especificação i

A prova de correção dos comandos de mudanças para uma especificação das mudanças pode ser demonstrada verificando que:

A(comandos(sistema i)) = mudanças(especificação i)

sendo que

comandos(sistema i) = sistema i+1

e

mudanças(especificação i) = especificação i+1

A tarefa do gerente de configuração seria realizar o mapeamento inverso ao da função A.

Outra propriedade importante neste processo de validação é a forma de expressar as condições para fazer reconfiguração de determinados componentes do sistema, sem ocasionar erros nos

outros. Na realidade, aparecem aqui duas fases a serem estudados em relação ao comportamento do sistema: uma é da execução das modificações, e a outra é depois de serem executadas as modificações. Este ponto, que é essencial para configuração dinâmica, está ainda na fase de discussões preliminares e será tratado com mais detalhe numa próxima seção deste capítulo.

Seria desejável poder testar o mapeamento entre a estrutura lógica e a física, para garantir que os recursos requeridos pelos módulos possam ser providos pelas estações físicas nas quais eles serão alocados. Para isto é necessário definir explicitamente os requisitos de recursos físicos na declaração dos módulos, como foi mencionado anteriormente.

Idealmente, deveria ser possível realizar alguma verificação semântica do comportamento da configuração, baseada na informação provida pelo comportamento de cada módulo. Algumas tentativas de especificar o comportamento lógico de cada módulo, e de definir métodos que possam combinar estes comportamentos para obter o comportamento total do sistema já foram feitas, como, por exemplo, por MILNER [112], mas as primitivas utilizadas são restritivas e a análise é difícil. Outro enfoque, apresentado por HOLENDERSKI em [69], pesquisa o uso de expressões regulares para especificar o comportamento dos processos como sequências de eventos. O comportamento do sistema (composição paralela) é dado por um operador que combina os comportamentos dos módulos, forçando a participação simultânea (interseção) nos eventos especificados como sendo de sincronização. Apesar de ter-se construido uma ferramenta simples para analisar a possibilidade de bloqueio perpétuo para sistemas pequenos, o poder deste enfoque é limitado. Esta é uma área na qual é preciso aprofundar as pesquisas ainda.

A verificação rigorosa de tipo das interfaces é relativamente fácil de implementar, e C/MESA e CONIC/C permitem fazê-la em tempo de configuração. Entretanto, não existe uma resposta simples para tratar da verificação semântica das configurações.

V.4.4 GERENCIAMENTO DA CONFIGURAÇÃO

Como já foi mencionado no início deste capítulo, o

gerenciamento da configuração dependerá fortemente da arquitetura física e lógica do ambiente, no qual será executado o sistema distribuido. Num enfoque do tipo máquina hospedeira/máquinas alvo a tendência é ter um gerenciamento mais centralizado tanto para a configuração estática quanto a dinâmica.

Na máquina hospedeira poderá ser interpretada e testada a correção da configuração, a ser carregada depois de forma distribuida no conjunto de máquinas alvo. No enfoque centralizado as mudanças também serão validadas na máquina hospedeira, que conterá toda a informação do sistema, para gerar os comandos que o sistema operacional deverá executar para implementar as mudanças necessárias. Desta forma, o gerenciamento da configuração estará concentrado na máquina hospedeira , deixando somente funções mínimas, correspondentes ao gerenciamento local de configuração, no núcleo de cada máquina alvo.

Um enfoque alternativo, menos centralizado, seria poder executar as mudanças de forma interativa em cada máquina alvo, dependendo somente de alguma informação geral que estaria na máquina hospedeira. Logicamente, para implementar este enfoque é necessário ter funções de gerenciamento mais poderosas no núcleo de cada máquina alvo.

Em parte estas idéias estão subjacentes na evolução do projeto CONIC apresentado nas últimas versões [105, 106], e que serão tratadas com mais detalhe na próxima seção, em relação à discussão sobre a unidade de configuração e de substituição.

Em [105], é mostrado como o suporte de execução de CONIC foi desenvolvido para facilitar a construção de sistemas distribuidos. O sistema é aberto, no sentido que as funções do suporte de execução estão disponíveis ao usuário, que pode ligar portas de seus módulos às das funções do suporte de execução. As estações lógicas podem executar tanto na máquina hospedeira com sistema operacional UNIX, quanto nas máquinas alvo sem sistema operacional. Para isto, deve-se acrescentar as funções mínimas necessárias ao nó lógico. Isto traz como consequência, que as partes críticas do sistema, em relação ao tempo, executem nas máquinas alvo, e que o gerenciamento dos componentes execute na máquina hospedeira, para usufruir das facilidades que esta última possui.

Em relação à maneira de realizar as mudanças na configuração

dinâmica, é fundamental satisfazer os requisitos de validação da especificação e de consistência do sistema, já analisados na seção anterior. A especificação das mudanças será traduzida para gerar comandos ao sistema operacional, precisando geralmente de informação sobre o seu estado. Esta informação poderá ser obtida a partir de um banco de dados mantido pelo gerenciador de configuração e/ou pelo sistema. O gerenciador pode consultá-lo para decidir que ações são necessárias, por exemplo, se um tipo de componente já está residente numa estação particular, ou se será necessário carregá-lo. Uma mudança simples pode resultar numa série de comandos, por exemplo, a criação de uma instância de um componente pode gerar uma consulta, um carregamento e uma instanciação.

Uma propriedade desejável, já mencionada anteriormente, é a de poder expressar as especificações das mudanças numa linguagem declarativa e não operacional. Desta forma não é definido como estas mudanças devem ser executadas. Isto dá mais liberdade ao gerenciador de configuração para decidir qual é a estratégia mais adequada para realizar as mudanças. Em particular, uma das propriedades mais importantes é minimizar o número de componentes que precisam ser interrompidos, para executar as mudanças. Por exemplo, numa substituição é possível diminuir o tempo de parada desses componentes, carregando e criando o novo módulo antes de destruir o antigo.

Se considerarmos como parte do processo de configuração a definição do mapeamento entre a configuração lógica e a configuração física, a precisão com que este é definido implicará numa maior ou menor liberdade em relação à estratégia de alocação. Quando a informação sobre a alocação não define exatamente em que estação deverá ser carregado cada módulo, existirá um determinado grau de liberdade para o gerenciador de configuração, que permitirá otimizar o uso dos recursos físicos do sistema de hardware, alocando os módulos em função da informação sobre os recursos necessários. Esta liberdade pode ser aproveitada também em situações de ocorrência de falhas, nas quais, se existir uma redundância suficiente no sistema, o gerenciador de configuração poderá realocar os módulos das estações que entraram em colapso, nas estações que estejam funcionando. Devemos notar que, neste caso, não ocorre mudança de

especificação de configuração.

V.4.5 UNIDADE DE CONFIGURAÇÃO E SUBSTITUIÇÃO

Como foi mencionado antes, na maioria dos ambientes de programação que oferecem facilidades para tratar configuração, ou através de primitivas embutidas na linguagem de programação, ou através de uma outra linguagem separada, a unidade de configuração é a unidade básica de estruturação da linguagem de programação. Mas devido à necessidade de poder estruturar sistemas grandes em níveis hierárquicos, algumas linguagens permitem definir outras unidades que são construidas a partir das unidades básicas.

V.4.5.1 LINGUAGEM CONIC

Como exemplo, analisemos o caso da linguagem de configuração CONIC/C, que, depois da sua primeira definição, teve várias modificações que ilustram a evolução da linguagem nos últimos anos.

Em [104] MAGEE apresenta uma versão da linguagem CONIC/C, na qual existe uma hierarquia de três níveis com interfaces idênticas, expressas por portas de entrada e saída. O nível mais baixo é o **módulo** ("module"), que contém uma ou mais **tarefas** ("tasks"); o nível intermediário é definido pela **estação** ("station"), que é composta por um ou mais módulos, e corresponde a uma unidade lógica que será carregada numa única unidade física; o terceiro nível é a **rede** ("network"), que está composta por várias estações lógicas que serão mapeadas em determinadas estações físicas. A primitiva de ligação **LINK** liga portas de módulos entre si, e portas de estações entre si. Os níveis em CONIC/C estão assim separados. A primitiva inversa **UNLINK** desfaz a ligação de portas de forma análoga. As primitivas de configuração dinâmica se aplicam também a módulos e estações, o que permite introduzir mudanças nestes dois tipos de unidades de configuração e substituição: os módulos e as estações.

Por ter achado esta hierarquização muito rígida, já que permite somente três níveis, foi introduzida por DULAY et alii [52] uma nova definição de estruturação. Foram definidos no nível mais baixo **módulos de tarefa** ("task module") que são as unidades

sequenciais de programa. No nível seguinte foram definidos os **módulos de grupo** ("group modules") constituidos de um ou mais módulos de tarefa ou módulos de grupo, permitindo um número arbitrário de níveis de estruturação. Foram definidos também **módulos de segmento** ("segment module"), que são módulos de grupo restritos a um mesmo espaço de endereçamento, isto é, os componentes de um módulo de segmento podem compartilhar código de procedimentos e funções, e podem ser passados ponteiros nas mensagens utilizadas na comunicação entre eles. As ligações são feitas entre portas de módulos de tarefa e entre portas de módulos de grupo. No último nível da hierarquia, o sistema é definido pela palavra reservada "system". Nesta versão, as unidades de configuração e substituição são os módulos de tarefa e os módulos de grupo de diferentes níveis de hierarquia.

Depois de variações intermediárias pouco significativas, na última versão [106] é definida como unidade de configuração o **nó lógico** ("logical node"). Uma aplicação distribuida consiste de um ou mais nós lógicos interligados, e cada nó lógico é composto por um conjunto de módulos de tarefa que executam pseudo concorrentemente. As tarefas pertencentes a um mesmo nó lógico compartilham o mesmo espaço de endereçamento. Num nó físico podem estar carregados um ou mais nós lógicos. Existem também os **módulos de grupo**, cuja implementação difere do nó lógico, em que este último inclui o suporte de execução. Um nó lógico pode ser executado numa máquina com sistema operacional UNIX, ou numa máquina alvo sem sistema operacional residente. Os autores afirmam que reconhecem o benefício de ter processos de baixo custo, para os quais os custos administrativos de criação e chaveamento são pequenos, como foi implementado por vários outros sistemas distribuidos, como por exemplo por CHERITON em [38] e por MULLENDER e TANENBAUM em [115].

O modelo inicial de configuração dinâmica estava baseado num banco de dados centralizado, que continha todas as informações sobre o estado do sistema. Para fazer um gerenciamento do sistema, robusto e distribuido, seria necessário implementar um banco de dados distribuido, encarando os problemas de realização de atualizações atômicas consistentes de dados replicados. Por achar esta solução complicada e de custo grande em relação ao tempo necessário para fazer reconfiguração, particularmente no

caso de acontecer uma falha, foi decidido mudar o modelo inicial.

Os autores do projeto CONIC consideram que para os requisitos dos usuários em relação à configuração dinâmica, deve ser satisfatório o gerenciamento no nível dos nós lógicos. O nó lógico será a unidade de configuração e a menor unidade de falha. Esta decisão reduz enormemente a quantidade de informação a ser manipulada pelo gerenciador do sistema. Por outro lado, em lugar de ter um banco de dados de configuração separado, foi decidido que uma aplicação teria seu próprio banco de dados. Cada nó lógico contém informação suficiente, para descrever a sua própria interface e as suas ligações com os outros nós. A quantidade de informação é suficientemente pequena para poder ficar na memória principal. O gerenciador de configuração obtém a informação sobre a aplicação consultando o servidor sobre os nomes do conjunto de nós lógicos que constituem a aplicação. A informação referente ao nó é obtida por comunicação direta com o nó.

Podemos notar a importância da definição da unidade de configuração e de substituição, pelas implicações que a sua escolha tem para a forma de gerenciamento da configuração.

V.4.5.2 LINGUAGEM ARGUS

Outro exemplo interessante de ser analisado é a linguagem ARGUS [99, 98, 97], para a qual foi discutido em [23] por BLOOM a substituição dinâmica de módulos num sistema de programação distribuida. ARGUS é uma linguagem integrada de programação e de desenvolvimento de sistemas, projetada para implementar aplicações distribuidas, com ênfase na confiabilidade e a consistência, mas que não tenham restrições severas de tempo real. A linguagem será analisada com mais detalhe no Capítulo VII, e portanto apresentamos aqui somente algumas características.

ARGUS é uma modificação e extensão de CLU (LISKOV et alii [95]), linguagem sequencial, com forte controle de tipo e que suporta abstração de dados. A diferença fundamental entre as duas reside na introdução em ARGUS, de guardiães ("guardians") e de ações atômicas.

Um guardião é um processador lógico, que encapsula algum recurso, com seu próprio estado, e processos internos múltiplos.

Ele provê manuseadores ("handlers"), que são objetos parecidos com procedimentos, que podem ter acesso à representação do guardião, mas que são invocados por chamadas remotas de procedimentos feitas por outros guardiães, para os quais foram exportadas as interfaces do guardião e dos manuseadores. Isto é realizado através do envio de mensagens para estações remotas.

As ações atômicas garantem que as falhas possam ser mascaradas e que a consistência seja mantida; elas, ou são executadas totalmente e com sucesso, ou são abortadas e a execução volta ao estado anterior ao início da ação. Qualquer ação, que possua a interface de um guardião, pode fazer chamadas remotas aos seus manuseadores.

Os guardiães podem ser criados dinamicamente por outros guardiães e podem também destruí-los. O sistema ARGUS inclui um serviço de catálogo que pode ser usado pelos clientes para localizar outros serviços. É importante salientar que em ARGUS as funções de criação e interconexão de guardiães não são independentes. Pelo contrário, quando um guardião cria outro guardião, ele deve definir a sua locação e os dois guardiães ficam aninhados.

Um sistema em ARGUS é formado por um conjunto de guardiães, que podem ser combinados em subsistemas, formando uma estrutura hierárquica. Além dos módulos do tipo guardião, existem em ARGUS outros tipos de módulos: agrupamentos ("clusters"), procedimentos e iteradores ("iterators"). Em [23] BLOOM discute qual é a unidade de substituição mais apropriada para dar suporte a configuração dinâmica, e qual é a informação necessária de ser mantida na substituição, para garantir a preservação das interfaces com os clientes.

Sendo o guardião a unidade básica de estruturação em ARGUS existem várias opções de escolha. Por um lado poderiam ser escolhidos os tipos de módulos contidos num guardião, tais como procedimentos, agrupamentos, manuseadores ou criadores ("creators"), para serem substituídos de forma independente. Outra opção seria escolher as abstrações compostas por um conjunto de guardiães que cooperam para implementar algum serviço.

Uma primeira decisão apresentada é a escolha do guardião como unidade mínima de substituição, justificada pelos seguintes

argumentos. Um guardião tem uma interface pequena e bem definida para o resto do sistema, descrita pelo conjunto de manuseadores que ele provê. Esta estrutura facilita a substituição do guardião sem afetar o resto do sistema. Outro argumento importante é a propriedade dos guardiães terem um estado explícito permanente. Dentro da definição de um guardião, é declarado como estável ("stable") o conjunto de objetos que são compartilhados por todos os manuseadores, e são mantidas cópias consistentes de seus estados, para garantir a sobrevivência do guardião em caso de falhas. Já que, neste último caso, é suposta a retomada de execução do guardião a partir desses dados, é razoável supor o mesmo comportamento no caso de substituição.

A substituição de módulos menores, tais como procedimentos ou agrupamentos, é mais trabalhosa, já que uma mesma implementação deles pode ser utilizada em diferentes guardiães, e deveriam ser todos eles identificados para essas implementações serem substituidas. Essas substituições podem ser consideradas casos particulares da substituição de um guardião. Entretanto, mesmo que as substituições de unidades menores fossem permitidas, de qualquer maneira as substituições de guardiães como unidades simples ainda seriam necessárias.

Por outro lado, é analisada a possibilidade de substituir uma unidade maior do que o guardião, chamada de subsistema e constituida por um conjunto de guardiães que cooperam para implementar um serviço, como já foi mencionado anteriormente. A linguagem ARGUS não reconhece as ligações entre os guardiães e não existe maneira de explicitar o estado dos guardiães como parte da implementação da abstração à qual eles pertencem.

A habilidade de substituir guardiães individuais não provê o poder de substituir um subsistema, sem restringir a substituição a preservar alguns dos detalhes da implementação do subsistema juntamente com a preservação da interface. O fato, de alguns dos manuseadores, providos pelos guardiães que compõem o subsistema, serem utilizados somente por outros componentes da instância do subsistema, é uma decisão da implementação. Outras implementações do subsistema podem usar outros tipos de componentes que provejam manuseadores diferentes. Estas decisões de implementação são sujeitas a mudanças durante o processo de substituição dinâmica. Se cada guardião é substituido individualmente, a sua interface

inteira deve ser preservada na substituição, já que não existe maneira de escolher apenas guardiães que usam alguns desses manuseadores para que sejam substituidos ao mesmo tempo. Portanto existem diferenças entre substituir o conjunto de guardiães que compõem um sistema, e substituir o subsistema como entidade única. Os outros subsistemas sabem que aquele foi o subsistema modificado e não precisam se preocupar com os guardiães. É possível também que alguns dos manuseadores de um determinado componente estejam disponíveis aos clientes, enquanto outros manuseadores sejam usados somente para comunicação dentro do subsistema. Em tais casos, os manuseadores visíveis aos clientes devem ser preservados, enquanto que o tipo do componente como um todo não precisa.

Infelizmente, ARGUS não pode distinguir entre guardiães que são usados unicamente dentro de um subsistema, e os que são disponíveis aos clientes. A falta de distinção, entre guardiães visíveis e não visíveis em um subsistema, apresenta um problema para o mecanismo de substituição. O princípio de separação entre interface e implementação sugere que as mudanças na implementação da abstração devam ser invisíveis aos clientes, e portanto que os detalhes de implementação possam ser sujeitos a modificações. Portanto o sistema de substituição deverá suportar a substituição de subsistemas como entidades simples, se for necessário garantir a consistência de tipo sem impor restrições excessivas nas formas de substituição permitidas.

Algum suporte explícito será necessário no sistema, para que o mecanismo de substituição possa distinguir entre os manuseadores que devem ser preservados e aqueles que podem ser eliminados. A decisão de BLOOM foi a de definir subsistemas na biblioteca de ARGUS, e a de manter nela as informações necessárias em relação às interfaces do subsistema em vez de acrescentar à linguagem ARGUS outro tipo novo de módulo, solução que seria bem mais complexa.

São discutidas depois as formas nas quais aparecem os subsistemas nos programas escritos em ARGUS, quais informações sobre as estruturas dos subsistemas precisam ser mantidas para uso do sistema de substituição, e como incorporar essa informação na biblioteca de ARGUS. É salientado que a razão principal pela qual foram examinados subsistemas no contexto de substituição,

foi devido à necessidade de escolher entre as propriedades de flexibilidade e segurança. Sem identificar subsistemas explicitamente, o sistema de substituição teria que permitir mudanças que não seriam seguras, ou proibir algumas mudanças que poderiam ser realizadas. Para maiores detalhes sobre esta discussão sugerimos consultar o trabalho de BLOOM [23].

Para finalizar este tema queremos destacar os pontos comuns entre esta última discussão, e a evolução da unidade de configuração e substituição mostrada anteriormente, do projeto CONIC. Existe uma tendência de não ficar somente preso à unidade de estruturação da linguagem de programação, e ir mais além analisando as vantagens de poder definir unidades de configuração e substituição de mais alto nível, que tenham um papel significativo para a estruturação hierárquica do sistema a ser desenvolvido.

V.5 CONDIÇÕES NECESSÁRIAS PARA RECONFIGURAÇÃO

V.5.1 DISCUSSÃO GERAL

No processo de reconfiguração dinâmica, deve ser considerada uma série de problemas que podem ocorrer e que precisam ser encarados, para evitar erros no funcionamento posterior do sistema. Procuraremos aqui levantar alguns deles, mas queremos salientar que só serão abordados os mais importantes, já que esta discussão está bem longe de estar esgotada, e, pelo estado da arte no tema, fica evidente que existe muita pesquisa para ser feita ainda.

Os casos mais comuns de reconfiguração de um sistema são os seguintes:

i) acrescentar algum módulo ao sistema já existente;

ii) tirar algum módulo do sistema;

iii) substituir um módulo de implementação por outro com implementação diferente da mesma interface.

Como estas modificações precisam ser feitas sem parar o sistema, é necessário tomar cuidado com as partes do sistema que serão evidentemente perturbadas por estas ações.

Por exemplo, no primeiro caso precisam serem feitas conexões novas com os módulos já existentes que se deseja ligar ao módulo novo. Esses módulos terão que ser interrompidos para poder executar essas novas conexões e, dependendo das funções que eles provêm, e dos estados nos quais eles se encontram, essa interrupção poderá ou não causar problemas. Se o módulo a ser acrescentado fica como intermediário entre dois ou mais módulos já ligados, além das novas conexões, é preciso destruir primeiro as conexões anteriores, e o problema causado pelas interrupções pode ser bem sério neste caso, já que envolve vários módulos e várias conexões que precisam ser destruidas.

No segundo caso os problemas são parecidos ao do caso anterior, no que se refere à necessidade de destruir conexões já existentes e ter que estabelecer novas, além de destruir o módulo retirado.

Nos dois casos, é necessário também atualizar a descrição do estado geral do sistema, para que o resto do sistema tenha conhecimento das mudanças realizadas. Dependendo da forma de gerenciamento da configuração, isto pode implicar em atualizar somente informação centralizada, ou em precisar replicar as atualizações em diferentes lugares do sistema.

No terceiro caso a principal verificação é em relação ao tipo de módulo que é substituido, que precisa ter a mesma interface que o anterior. Existe ainda o problema de interrupção que deve ser levado em conta para escolher o momento adequado para fazer a substituição. Será visto mais adiante que em determinados casos pode até ser necessário modificar a interface, além da implementação, o que ocasionaria uma série de problemas suplementares que serão discutidos.

Existe ainda a preocupação sobre o estado do sistema durante a reconfiguração, e o problema da sua consistência em relação ao resto do sistema que continua funcionando.

Na linguagem ARGUS parte destes problemas são encarados e resolvidos através da implementação de ações atômicas e da definição e manutenção de um conjunto de objetos do guardião declarados como estáveis ("stables"), que salvam o estado do guardião no caso de interrupção ou falha, como já foi mencionado anteriormente.

As ações atômicas garantem que as falhas ou interrupções

sejam mascaradas e a consistência mantida. Como já foi mencionado anteriormente, a execução de uma ação tem dois resultados possíveis: ela é completada, e neste caso as mudanças são validadas, ou ela é abortada e as mudanças anuladas. Em ARGUS as ações podem ser aninhadas: uma ação pode conter sub-ações que podem executar determinado código e podem completar ou abortar sem completar ou abortar a ação do pai. Mas o aborto da ação de um pai anula todas as mudanças realizadas por todas as suas sub-ações, mesmo as que terminaram. O estado estável de um guardião só é modificado quando a ação de nível mais alto (ou a raiz da árvore de ações aninhadas) é completada.

Podemos notar que a linguagem ARGUS foi desenvolvida tendo como preocupação principal a confiabilidade do sistema a ser construido.

No caso do projeto CONIC, a preocupação com a consistência do sistema no processo de reconfiguração não tinha sido considerada nas primeiras versões, e somente à partir de 1986 em [105] foram feitas algumas propostas que evoluiram em versões posteriores, para enfrentar parte dos problemas.

Em [105], na apresentação da linguagem de configuração, foi introduzida uma nova declaração chamada "transaction", que é utilizada tanto para envolver as declarações que definem uma configuração inicial, quanto as declarações que definem as mudanças a serem realizadas sobre uma configuração existente. A declaração é definida da seguinte forma: ela garante que a lista de declarações seja totalmente processada, ou se acontecer algum problema durante o processamento seu efeito seja equivalente ao da lista não ter sido processada. Entretanto, é notado que se a transação destruir uma tarefa, a informação sobre seu estado interno será perdida. O gerenciador da configuração primeiro confere que as declarações da transação são válidas, testando que as imagens das estações lógicas estão disponíveis e são do tipo correto para o alvo especificado, e que as ligações são feitas entre portas de entrada e de saída do mesmo tipo. Só depois é que serão processadas as declarações. Se uma declaração falhar, as declarações previamente processadas na transação serão anuladas. As estações criadas serão destruidas e as ligações desfeitas usando as operações que são disponíveis também aos usuários. Isto significa que as falhas de configuração não resultarão em

sistemas parcialmente construídos, e que as mudanças produzirão configurações consistentes.

Na última versão de CONIC apresentada em [106], é levantada a preocupação com a possibilidade de interrupção de um módulo pelo processo de configuração dinâmica. A solução dada a este problema é ilustrada através da programação de um exemplo, no qual o módulo tarefa precisa estar ciente do perigo de interrupção. É transferida para o programador do módulo tarefa a responsabilidade de colocar as condições de sincronização, para não permitir interrupções durante o processamento que não pode ser interrompido. Essas condições precisam então ser incorporadas na lógica do módulo tarefa, e elas se encarregarão de prover a sincronização com o gerente de configuração do sistema.

Apesar de insistir na importância de que os módulos do sistema sejam projetados e programados de maneira independente em relação ao ambiente no qual serão executados, os autores de CONIC apontam a necessidade do programador estar ciente da configuração dinâmica. Isto é consequência do fato de que a informação, necessária para saber quando é seguro realizar mudanças, está embutida nos módulos da aplicação. Os autores reconhecem que, mesmo tendo comprovado que esta solução é satisfatória, do ponto de vista de programação ela é tosca, e é necessário procurar soluções mais elegantes. Curiosamente, nesta versão não há nenhuma referência à declaração "transaction" da versão anterior, e portanto não conseguimos concluir se ela foi abandonada e por quais razões.

No Capítulo VII apresentaremos algumas opiniões nossas sobre estas últimas versões de CONIC, e em particular sobre as soluções sugeridas para os problemas levantados na configuração dinâmica.

V.5.2 DISCUSSÃO SOBRE SUBSTITUIÇÃO BASEADA NA LINGUAGEM ARGUS

Achamos interessante apresentar a seguir a discussão feita por BLOOM em [23] sobre a legalidade da substituição de uma instância de um subsistema por outra. Lembremos a discussão apresentada numa seção anterior sobre a conveniência de considerar como unidade de substituição o subsistema para a linguagem ARGUS, composto por um conjunto de módulos (ou guardiães) que cooperam para implementar uma determinada função.

Se o sistema ARGUS permitir substituição dinâmica sem alterar a modularidade e a confiabilidade do sistema, deve ser garantido que os módulos possam continuar dependendo da funcionalidade correta dos módulos que eles utilizam, independentemente do fato de serem realizadas modificações neles.

No caso dos módulos é intuitivo considerar que se dois módulos satisfazem a mesma especificação de interface, um pode ser utilizado no lugar do outro. Quando for necessária uma dada abstração, qualquer implementação da abstração pode ser utilizada. Nesta afirmação BLOOM esclarece que não distingue entre um subsistema formado por um ou mais guardiães. Entretanto, ela coloca que quando se considera a substituição dinâmica, é mais difícil determinar exatamente o que é que esse critério intuitivo significa e quando ele é satisfeito. No entanto, mantendo o princípio de satisfazer a mesma especificação de interface, em determinados casos, uma implementação de um subsistema em ARGUS correspondente a uma abstração não pode ser utilizada para substituir outra implementação da mesma abstração. Por outro lado, uma implementação de uma abstração diferente pode comportar-se apropriadamente se for substituir a instância corrente. Estes conceitos serão ilustrados por exemplos apresentados mais adiante.

São então levantados dois problemas a serem analisados. O primeiro consiste em poder definir quando uma abstração pode ser utilizada para substituir outra. O segundo consiste em poder determinar quando uma dada implementação de uma abstração pode ser utilizada para substituir a instância corrente.

Substituição da abstração

A definição mais simples de substituição é aquela que exige que as abstrações não possam mudar, isto é, que somente as implementações de uma determinada abstração possam mudar durante a substituição dinâmica. Entretanto, existem casos nos quais este critério pode ser relaxado, enquanto for garantido que o comportamento do módulo não mude de forma visível aos clientes. Existem casos também, nos quais satisfazer a abstração inicial não é suficiente para garantir que a nova instância possa substituir a existente da mesma abstração.

BLOOM aponta o interesse em relaxar a restrição em dois

casos específicos. O primeiro é o caso no qual é projetada uma nova abstração específica para substituir as instâncias já existentes, e que tem um comportamento aceitável para continuar a partir de determinado estado não inicial, mas que não pode tratar de situações que poderiam ter surgido antes da substituição. O segundo caso é aquele no qual a abstração inicial não contém algumas operações que serão necessárias no futuro. Neste caso é desejável substituir a instância por outra instância da **abstração estendida**, que permita que os clientes já existentes continuem utilizando o módulo, enquanto proveja a novos clientes o conjunto estendido de operações. Existem casos também, nos quais satisfazer a abstração inicial não é suficiente para garantir que a nova instância possa substituir a existente da mesma abstração.

Será mostrado a seguir como o requisito de manter a mesma abstração é às vezes insuficiente, e quando podem ser substituidas abstrações com a mesma interface.

Para ilustrar a discussão é apresentado o exemplo do gerador de identificadores únicos. A especificação da abstração estabelece que não serão repetidos identificadores durante um númeto grande X de invocações da operação que gera os identificadores. Supondo que, depois de terem sido gerados N identificadores (N<X), se deseje substituir a instância corrente por uma nova implementação, por exemplo, com um algoritmo mais eficiente; quais seriam as especificações que deveriam ser satisfeitas pela substituição?

Substituir a nova implementação da abstração satisfazendo somente as especificações originais, neste caso, não seria mais suficiente. Existem outras condições, que precisam ser satisfeitas: por um lado não podem ser repetidos os N identificadores que já foram gerados, e, para garantir as especificações originais, é necessário ainda gerar X-N identificadores diferentes. Portanto, existem **condições de continuação** que precisam ser satisfeitas, além das originais. Por outro lado, deve ser garantida a propriedade de que qualquer implementação de uma abstração deve poder substituir ela mesma, já que uma instância pode ser interrompida em qualquer momento e ser substituida por uma cópia exata dela mesma.

Este exemplo sugere a necessidade de um estudo mais detalhado sobre alguns temas, que precisam ser considerados na

definição de substituição dinâmica. É necessário achar uma maneira de derivar as especificações originais do módulo, e definir o que significa para uma implementação satisfazer as especificações de continuação.

Por outro lado, é destacado em [23] que as noções de abstração e especificação mudam quando são introduzidas as abstrações de continuação. Sem substituição dinâmica, havia uma suposição implícita de que as instâncias eram sempre iniciadas num estado inicial definido pelas operações de criação. A abstração de continuação pressupõe que a instância é iniciada com um estado definido em parte pela outra instância, e em parte pelo procedimento de substituição. A especificação do comportamento da abstração depende dos eventos prévios já ocorridos.

Substituição de instância

Para que uma instância da abstração i possa ser substituida por uma instância da abstração j, existem condições de correção sobre a instância da abstração j que devem ser verificadas mesmo sabendo que a abstração j satisfaz os critérios de correção para substituição de abstração. Estas condições envolvem principalmente a correspondência entre o estado final da instância substituida e o estado de partida da instância substitutiva. Como o formato do estado da instância substituida depende da implementação, é necessário ter mecanismos para preservar o valor do estado.

Extensão das abstrações

Na discussão anterior sobre as abstrações de continuação era pressuposto que a interface das abstrações não era modificada. No caso dos subsistemas era pressuposto que o conjunto de manipuladores não mudava. Será analisado agora em que circunstâncias é permitido mudar as interfaces, assim como as implementações de um subsistema, durante a substituição dinâmica.

As mudanças na interface podem ser consideradas como combinações de adição de novos manuseadores e de eliminação de alguns já existentes. Em [23] são considerados fundamentalmente as mudanças que envolvem somente adições de manuseadores. A eliminação de manuseadores não cria problemas, somente no caso de

nenhum dos clientes já existentes utilizá-los.

A visibilidade de uma extensão para os clientes já existentes, dependerá dos tipos de propriedades que poderão ser especificados sobre as abstrações, e dos tipos de afirmações que os clientes poderão fazer, baseados nessas especificações. São apresentadas em [23], de modo informal, algumas classes de extensões, e dadas explicações intuitivas de quando essas extensões satisfazem certos critérios de segurança, de maneira que os usuários que desejem estender um subsistema tenham critérios para avaliar a segurança de tais ações. São consideradas três classes de extensões que apresentaremos a seguir.

A primeira classe de extensões consiste na adição de operações de consulta. Como as consultas são do tipo leitura somente, elas não provocam nenhum efeito sobre o estado do subsistema. Um exemplo deste tipo de extensão poderia ser a adição de um manuseador para o sistema de correio (muito utilizado na literatura e em particular em [23]) da forma abaixo.

lista correio (nome do usuário)

Os clientes com acesso unicamente ao conjunto original restrito de operações não perceberiam nenhuma mudança com esta extensão. Entretanto, os clientes poderão enxergar agora mais informação além daquela que eles já enxergavam antes. Se a linguagem assertiva, associada com a especificação original, permitisse ao cliente deduzir que a informação adicional era invisível, este tipo de extensão também não seria segura. Outro exemplo de uma assertiva, que poderia ser violada pela adição de operações de consulta, é a assertiva de que um determinado evento ocorre somente como continuação de um outro evento. A introdução de uma operação de consulta pode mudar essa ordenação.

A segunda classe de extensões é a adição de operações de atualização, mas que não afetam nenhum valor visível através do conjunto original de operações. A propriedade principal desta classe é que, apesar das novas operações atualizarem o estado, as extensões permanecem invisíveis aos clientes que não usam as operações acrescentadas. Estas extensões consistem comumente em adicionar componentes ao estado corrente e na adição de manuseadores que operam somente sobre estes componentes.

Novamente neste caso, se fosse possível escrever assertivas sobre propriedades, sem ser sobre os valores retornados por várias invocações, não poderíamos garantir a invisibilidade de tal extensão. Estas duas classes de extensões fazem parte da categoria denominada de **extensões sem interferência.**

A terceira classe de extensões é aquela para a qual é difícil garantir que as assertivas, sobre as quais dependem os clientes existentes, não sejam violadas. Esta classe inclui várias extensões que são muito úteis, e se caracterizam pela adição de manuseadores que interagem com os manuseadores originais através dos dados do subsistema. A propriedade significativa desta classe é que, se um cliente usa os manuseadores novos, a extensão pode ser visível aos clientes que usam unicamente o conjunto original de manuseadores.

Os problemas levantados em relação à substituição dinâmica devem ser encarados em qualquer sistema que pretenda suportar essa flexibilidade garantindo certo grau de segurança, usando uma linguagem com forte controle de tipo.

A análise em [23] prossegue formalizando as condições de correção para substituição dinâmica, ligados mais particularmente ao caso de subsistemas na linguagem ARGUS, que ajuda a esclarecer alguns problemas semânticos da substituição. Em particular, é discutido o problema da incompatibilidade de implementações, as restrições de comportamento por substituições sucessivas, e as substituições de abstrações em ARGUS.

Nos capítulos seguintes são tratados os problemas em relação aos requisitos do usuário, para localizar as instâncias de subsistemas, e é apresentado um mecanismo específico para dar suporte à substituição na linguagem ARGUS. Esse mecanismo consiste num conjunto pequeno e simples de comandos, para definir os tipos dos subsistemas a serem substituidos, para eliminar guardiães, para recuperar e instalar estados estáveis e para finalizar a substituição. A parte mais complexa do sistema de substituição é a que deve dar suporte ao ambiente do usuário. Uma das dificuldades principais no desenvolvimento do mecanismo de substituição é causada pela necessidade de manipular tipos e implementações de tipos explicitamente, o que a linguagem ARGUS não reconhece. Para reconhecer os subsistemas como entidades sem mudar a linguagem ARGUS e sem comprometer a segurança de tipo,

foi necessário estender as funções da biblioteca.

BLOOM coloca que os comandos apresentados são suficientes para expressar as substituições necessárias, e para descrever a semântica da substituição no sistema ARGUS, mas que eles são insuficientes como serviço a nível de usuário. Ela sugere que é importante desenvolver uma metodologia para realizar as substituições, e uma linguagem para dar suporte a essa metodologia que preveja o usuário com uma visão de alto nível da substituição e ajude a expressar as tarefas de substituição de forma precisa.

CAPÍTULO VI

DEFINIÇÃO E IMPLEMENTAÇÃO DE UMA LINGUAGEM DE CONFIGURAÇÃO PARA O AMBIENTE DE PROGRAMAÇÃO DISTRIBUIDA BASEADO EM MODULA-2

No capítulo anterior foram apresentados os conceitos principais sobre a necessidade de configuração de um sistema modular e em particular de um sistema distribuido. Foram discutidas também as diferentes maneiras de definir e implementar a configuração de um sistema, mostrando especificamente as vantagens de ter uma linguagem de configuração separada da linguagem de programação dos módulos que compõem o sistema. Esta análise norteou nosso trabalho e nos levou a definir uma linguagem de configuração estática a ser acrescentada à Modula-2, com as extensões já apresentadas para comunicação e sincronização, para construir o ambiente de programação distribuida desejado.

Neste capítulo apresentaremos a definição da linguagem mostrando as principais características, ilustradas através de um exemplo introduzido por STEVENS et alii [158], e outros que são extensões deste. Será descrita a seguir a implementação do interpretador desenvolvido por VETROMILLE em [170] para processar a linguagem de configuração estática, fase necessária para depois carregar e inicializar o sistema antes de começar a sua execução.

VI.1 DESCRIÇÃO DA LINGUAGEM DE CONFIGURAÇÃO

A primeira extensão a ser introduzida para poder definir a configuração de um sistema distribuido é o conceito de **tipo de módulo**, que em Modula-2 não existe. Este conceito é necessário para poder criar várias instâncias de um mesmo módulo em estações distintas, ou até na mesma estação.

Instâncias de módulos são obtidas a partir de tipos de módulos, guardados num sistema de armazenamento em memória secundária (sistema de arquivos). A linguagem Modula-2 não define tipos de módulos, mas podemos suprir esta falta facilmente da seguinte maneira, sem alterar a sintaxe da linguagem. Modula-2 requer que o nome de um módulo de implementação seja o mesmo que o do módulo de definição correspondente. Convencionalmente,

guardamos o módulo de implementação num arquivo com este mesmo nome. Para alcançar nossas metas, permitimos que os nomes dos arquivos contendo o código objeto de um módulo de implementação sejam diferentes do nome usado na sua declaração em Modula-2. Logo a um módulo de implementação associamos dois nomes: um que corresponde ao módulo de definição (sua interface) e outro que corresponde ao arquivo que o contém. Este segundo nome será utilizado como o nome do tipo de módulo para criar várias instâncias dele.

Para permitir o uso de múltiplas instâncias de um tipo de módulo de implementação, criamos um segmento de dados próprio para cada instância. Frisamos que pode ser usada a mesma cópia do segmento de código por todas as instâncias de um tipo de módulo carregadas na mesma estação.

Devido a modularidade de Modula-2, que permite definir um módulo separando a sua interface da sua implementação, incluímos também a possibilidade de definir **diferentes módulos de implementação para uma mesma interface**, utilizando o conceito anterior de tipo. Desta forma, para uma mesma definição de interface de módulo, podemos criar diferentes tipos de módulos de implementação, dos quais podem depois ser criadas várias instâncias. Cada tipo de módulo terá um nome diferente e estará associado a ele o nome da interface correspondente; no código gerado para o módulo de implementação será incluído o nome da interface. Estes conceitos serão ilustrados na próxima seção através de exemplos que ajudarão a perceber o potencial da linguagem.

A partir destes conceitos podemos deduzir que a linguagem de configuração considera como **unidade de configuração e substituição o módulo global de Modula-2.**

Outros conceitos utilizados na configuração são os de estação lógica e estação física. Uma **estação lógica** é composta por um conjunto de módulos ligados que formam uma unidade lógica a ser carregada numa **estação física**. Numa estação física podem estar carregadas uma ou mais estações lógicas. A linguagem de configuração proposta permite descrever configurações hierárquicas; isto significa que ela define configurações compostas por módulos programados em Modula-2 ou por (sub-) configurações já definidas anteriormente. Tanto os módulos quanto

as configurações são tipos, a partir dos quais podem ser criadas várias instâncias. É importante destacar que as (sub-) configurações estão formadas unicamente de módulos ou sub-configurações, e não podem ser combinados módulos com sub-configurações numa mesma (sub-) configuração. Portanto o primeiro nível da hierarquia da configuração será sempre composto exclusivamente por módulos programados em Modula-2 e os níveis seguintes por sub-configurações.

Descreveremos a seguir a especificação da linguagem de configuração que desejamos implementar para montar o ambiente distribuido baseado na linguagem Modula-2. A sintaxe e a semântica da linguagem de configuração serão descritas de maneira informal, e na seção seguinte serão apresentados exemplos que ajudarão a ilustrar seu uso e alcance.

A linguagem de configuração é uma linguagem declarativa, que, a partir dos módulos escritos em Modula-2, define uma configuração de forma hierárquica, e será interpretada por um programa, também escrito em Modula-2. Para completar a fase de configuração, temos acrescentado à linguagem uma diretiva que carrega a configuração lógica interpretada numa configuração física.

A sintaxe da definição de uma configuração se inicia com a seguinte declaração:

CONFIGURATION: <Identificador do tipo da configuração>;
por exemplo:

CONFIGURATION: PatientConf;

define um tipo de configuração com nome PatientConf.

A declaração seguinte tem como objetivo definir o contexto da configuração. Para isto é necessário declarar todos os tipos de módulos ou de sub-configurações que serão utilizados dentro da (sub-) configuração para criar instâncias, e os tipos das interfaces importadas. As instâncias de tipos de módulos criados dentro da (sub-) configuração correspondem a módulos de implementação. Para não complicar a apresentação da linguagem, queremos esclarecer que nesta seção utilizaremos nomes iguais para as interfaces dos módulos e para os tipos de módulos de

implementação. Isto corresponde ao caso particular no qual só existe um tipo de módulo de implementação para um determinado tipo de módulo de interface. Os outros casos, como já foi mencionado antes, serão apresentados nas próximas seções através de exemplos.

A sintaxe da declaração **USE** é a seguinte:

USE <lista de tipos de módulos de implementação ou sub-configurações e lista de interfaces de módulos>;

por exemplo:

```
USE kwind, psim, palarm;
USE PatientConf, NurseConf;
```

Cada (sub-) configuração é composta por módulos ou sub-configurações que pertencem a determinadas estações lógicas. Essas estações devem ser declaradas da seguinte forma:

STATIONS <lista de estações lógicas>;

por exemplo:

```
STATIONS Bed, Nurse;
```

A lista de estações pode conter um ou mais elementos, já que uma configuração pode estar composta por módulos ou sub-configurações que pertencem a uma ou mais estações lógicas.

Em Modula-2 os módulos interagem através de interfaces bem definidas, geralmente compostas de listas de procedimentos exportados. Estes serão chamados por outros módulos que precisam importar explicitamente as interfaces que os contêm.

No nível de configuração de um sistema composto de módulos escritos em Modula-2, é então necessário associar esses procedimentos às instâncias adequadas dos módulos criados. As ligações dentro das configurações e entre diferentes configurações são feitas por uma declaração que será apresentada mais adiante. Mas para poder fazer estas ligações é necessário declarar explicitamente importações de interfaces de módulos e exportações de instâncias de módulos de implementação, depois da declaração **STATIONS**. Estas declarações são opcionais, isto é, nem todas as configurações importam interfaces de módulos ou exportam instâncias de módulos de implementação, por exemplo, a

configuração de mais alto nível, numa estrutura hierárquica, pode não importar interfaces de módulos nem exportar instâncias de módulos de implementação.

A sintaxe destas declarações é a seguinte:

IMPORT <lista de interfaces de módulos>;

A lista de interfaces de módulos tem a seguinte sintaxe, na qual as chaves indicam a possível repetição:

{ <nome local de interface de módulo> : <tipo da interface importada> ; }

EXPORT <lista de instâncias de módulos de implementação>;

A lista de instâncias de módulos de implementação tem a seguinte sintaxe:

{ <nome local do módulo de implementação> : <tipo do módulo de implementação> ; }

A declaração indica, no caso de importação, o nome da interface do módulo e seu tipo, e, no caso de exportação, o nome do módulo de implementação e seu tipo. Neste último caso o módulo de implementação corresponde à instância criada na configuração à qual pertence a declaração de exportação. Na importação, o tipo de interface de módulo corresponde a um módulo pertencente a outra configuração, e o nome usado tem as características de um parâmetro formal do módulo que declara a importação. Esclarecemos novamente que neste exemplo os nomes dos tipos coincidem nas duas declarações, de importação e exportação, mas em geral, não será assim já que, quando existir mais de um tipo de módulo de implementação para uma mesma interface, os nomes serão necessariamente diferentes.

Esta explicação corresponde ao primeiro nível na hierarquia de configuração. Nos níveis seguintes, se as importações e exportações não casarem dentro de uma sub-configuração, elas precisam ser propagadas e serão associadas às sub-configurações ás quais pertencem, repetindo as declarações de importação e exportação correspondentes. Para estes casos existem regras de escopo para a visibilidade das interfaces de módulos importados e dos módulos de implementação exportados. Precisa-se tomar cuidado de como repetir as declarações para não violar a visibilidade

desejada. Desta forma, nas configurações aninhadas, na hora de fazer as ligações com os módulos exportados, estes serão qualificados unicamente pelo último nível no qual foi feita a declaração, não precisando explicitar a cadeia de sub-configurações à qual pertencem. Estes conceitos serão ilustrados através de exemplos na próxima seção.

Quando uma mesma (sub-) configuração precisa declarar importações e exportações, esta ordem das declarações deve ser respeitada, já que em alguns casos as instâncias dos módulos que são exportados podem utilizar alguma informação das interfaces de módulos importados, como acontece de forma análoga no nível dos módulos de Modula-2 em relação à importação e exportação de interfaces.

Por exemplo, em configurações diferentes poderiam estar as duas declarações seguintes:

Na configuração NurseConf:

```
EXPORT Selector : psel;
       ConsoleNurse : zterm;
```

e na configuração PatientConf:

```
IMPORT Select : psel;
       ConsoleN : zterm;
```

Numa fase posterior, podemos então ligar uma ou mais instâncias de PatientConf a uma instância de NurseConf, casando assim as listas de exportação e importação.

A próxima declaração define as instâncias de módulos ou de sub-configurações que compõem a configuração, explicitando também a alocação lógica.

A sintaxe da declaração **CREATE** é a seguinte:

CREATE <lista de criação de instâncias de módulos de implementação>;

ou

CREATE <lista de criação de instâncias de sub-configurações>;

A lista de criação de instâncias de módulos é composta por declarações com a seguinte sintaxe:

{ <um ou mais nomes de módulos> : <tipo do módulo implementação> **ON** <nome de estação lógica> ; }

por exemplo:

```
CREATE
   ErrorWindow, Window : kwind ON Bed;
   Scanner : psim ON Bed;
   AlarmN : palarm ON Nurse;
```

A declaração CREATE cria várias instâncias de módulos de implementação de tipos definidos (por exemplo, a primeira declaração cria duas instâncias do mesmo tipo), indicando em todos os casos a sua alocação lógica.

A lista de criação de instâncias de sub-configurações é composta por declarações com a seguinte sintaxe:

{ <um ou mais nomes de configurações> : <tipo da configuração> **ON** <lista de estações lógicas> ; }

A lista de estações lógicas deve conter as estações na ordem e número correspondentes às estações declaradas anteriormente na declaração **STATIONS**, na declaração do tipo da configuração.

por exemplo:

```
CREATE
   Maria : NurseConf ON Nurse;
   Pat1  : PatientConf ON Bed1, Nurse;
```

A configuração do tipo PatientConf foi declarada com duas estações lógicas; por isto, na hora de criar instâncias de configurações desse tipo, precisa-se associar a cada instância duas estações lógicas.

Depois de ter definido as instâncias de módulos ou sub-configurações que compõem a configuração, é necessário estabelecer as ligações entre importadores e exportadores. Isto é realizado através da declaração **LINK** que tem o seguinte formato:

LINK <lista de ligações de instâncias de módulos>;
ou
LINK <lista de ligações de instâncias de sub-configurações>;

A lista de ligações de instâncias de módulos está composta por declarações com a seguinte sintaxe:

{ <um ou mais nomes de instâncias de módulos> **WITH** <um ou mais nomes de instâncias de módulos>; }

Esta declaração liga uma instância de cada um dos módulos importadores à esquerda do **WITH** com as instâncias dos módulos

exportadores indicadas à direita do WITH. Estas últimas são os exportadores das interfaces que são importadas pelas instâncias de módulos especificadas à esquerda do WITH. É aqui então que é realizada a associação entre a interface do módulo importada com a instância do tipo do módulo de implementação associada a essa interface, além de definir as ligações das instâncias.

A associação de exportadores com as interfaces importadas utiliza o modelo posicional, isto é, a lista de nomes de instâncias de módulos à direita do WITH deve corresponder em número, ordem e tipo à lista de importações de módulos declarada dentro do tipo de módulo sendo ligado. Queremos salientar aqui que, como em Modula-2 as declarações de importações de um módulo podem estar tanto no Definition Module quanto no Implementation Module, a lista de importações de um módulo será composta pelas importações declaradas no Definition Module, seguidas das declaradas no Implementation Module, quando existirem. Quando as instâncias de módulos à esquerda do WITH tiverem listas de importações que correspondem à listas de instâncias de módulos à direita que são idênticas, estas poderão ser combinadas, colocando uma lista dos nomes das instâncias importadoras à esquerda do WITH.

As instâncias dos módulos nomeadas depois da palavra WITH podem ou não estar na mesma estação lógica ou configuração que as instâncias dos módulos nomeados à esquerda da palavra WITH. Dependendo da alocação física das instâncias dos módulos, estas ligações podem ser locais ou remotas. Isto será definido no final da configuração quando, for especificado o mapeamento entre a configuração lógica e a configuração física.

Exemplo:

```
LINK
   Command  WITH  Monitor, Console;
   Window   WITH  Console;
   Monitor  WITH  Select, Alarm;
```

A consistência entre as declarações de importações e exportações ao nível dos módulos já foi feita na compilação dos módulos. O interpretador da linguagem de configuração precisa então só conferir que, nas instâncias nomeadas à esquerda do WITH, as importações de interfaces de tipos de módulos correspondem com os tipos das instâncias de módulos de

implementação à direita do **WITH**.

A lista de ligações de instâncias de sub-configurações é composta por declarações com a seguinte sintaxe:

{ <um ou mais nomes de instâncias de sub-configurações> **WITH** <um ou mais nomes de instâncias de módulos qualificados> ; }

<nome de instância de módulo qualificado> ::= <nome de instância de sub-configuração> . <nome de instância de módulo de implementação>

Esta declaração liga uma instância de cada uma das sub-configurações, que aparecem à esquerda do **WITH**, a uma ou mais instâncias de tipos de módulos de implementação, que serão determinadas através da identificação da hierarquia de configurações à qual pertencem. Como já foi mencionado anteriormente, pela propagação das exportações a instância do módulo será qualificada somente pelo último nível de sub-configuração que contém a sua declaração de exportação.

Exemplo:

```
LINK
   Pat1, Pat2 WITH Maria.Selector, Maria.ConsoleNurse;
```

Cabe ao interpretador verificar que as instâncias dos módulos citados como exportadores nas declarações **LINK** realmente sejam compatíveis com as interfaces importadas pelas instâncias de módulos e sub-configurações importadoras.

Por causa de não poder combinar módulos e sub-configurações numa mesma (sub-) configuração, pode surgir a situação de precisar declarar uma sub-configuração contendo uma única instância de módulo, e neste caso não haverá declaração **LINK**; portanto esta declaração também é opcional.

Para ter uma idéia global da definição de configuração podemos mostrar o seguinte esquema informal. Uma declaração de configuração será definida como ilustra a Figura (VI.1).

```
CONFIGURATION: <nome do tipo de configuração>;
   USE  <lista de tipos de módulos de implementação ou sub-
         configurações e lista de interfaces de módulos>;
   STATIONS <lista de estações lógicas>;
   [IMPORT <lista de interfaces de módulos>];
   [EXPORT  <lista de instâncias de módulos de implementação)];
   CREATE   <lista de criação de instâncias de módulos de
         implementação ou de instâncias de sub-configurações>;
   [LINK    <lista de ligações de instâncias de módulos ou de
         instâncias de sub-configurações>];
END <nome do tipo de configuração>.
```

FIGURA VI.1 - DEFINIÇÃO DE DECLARAÇÃO DE CONFIGURAÇÃO

O último passo para concluir a fase de configuração é especificar o mapeamento entre a configuração lógica e a configuração física, ou seja, definir a alocação física dos módulos que compõem a configuração. Isto pode ser feito usando a seguinte diretiva:

```
LOAD  <nome da instância de configuração>  :  <nome do tipo de configuração>
                ON    <lista de estações físicas>.
```

A lista de estações físicas pode conter nomes simbólicos ou endereços físicos, mas ela deve corresponder em número e ordem à lista de estações declaradas na definição da configuração.

Exemplo:

```
LOAD  Ward : WardConf ON  Station1, Station2, Station3, Station4,
                          Station5, Station6.
```

Esta declaração será utilizada para o nível mais alto da hierarquia de configuração, isto é, depois de definida a configuração total do sistema distribuido. Queremos esclarecer que o nível mais alto será representado, na maioria dos casos, por um único tipo de configuração englobando todas as partes que o compõem. Mas pode também estar formado por mais de um tipo de configuração, permitindo deixar algumas ligações soltas de maneira a serem combinadas depois com outras partes necessárias ao sistema.

VI.2 EXEMPLOS ILUSTRATIVOS

VI.2.1 EXEMPLO DA ENFERMARIA

Vamos agora apresentar o exemplo escolhido para ser

especificado na nossa proposta de linguagem. O exemplo introduzido por STEVENS et alii [158] é o seguinte:

"Um hospital precisa de um sistema de controle de pacientes. Cada paciente é controlado por um dispositivo análogo que mede fatores tais como pulso, temperatura, pressão do sangue e resistência da pele. O sistema lê esses fatores periodicamente. Para cada paciente são especificados limites de variação de segurança para cada fator. Se um fator tomar um valor fora do limite de variação definido ou se um instrumento falhar, a estação da enfermaria é notificada".

Este exemplo já foi desenvolvido na linguagem CONIC (MAGEE [104], e nós o reestruturamos para poder ser escrito na linguagem Modula-2, usando a linguagem de configuração proposta.

Este projeto decompõe a função do sistema em sub-funções que podem ser associadas a módulos exportando e importando interfaces específicas.

O esquema físico do sistema de controle pode ser representado pela Figura (VI.2):

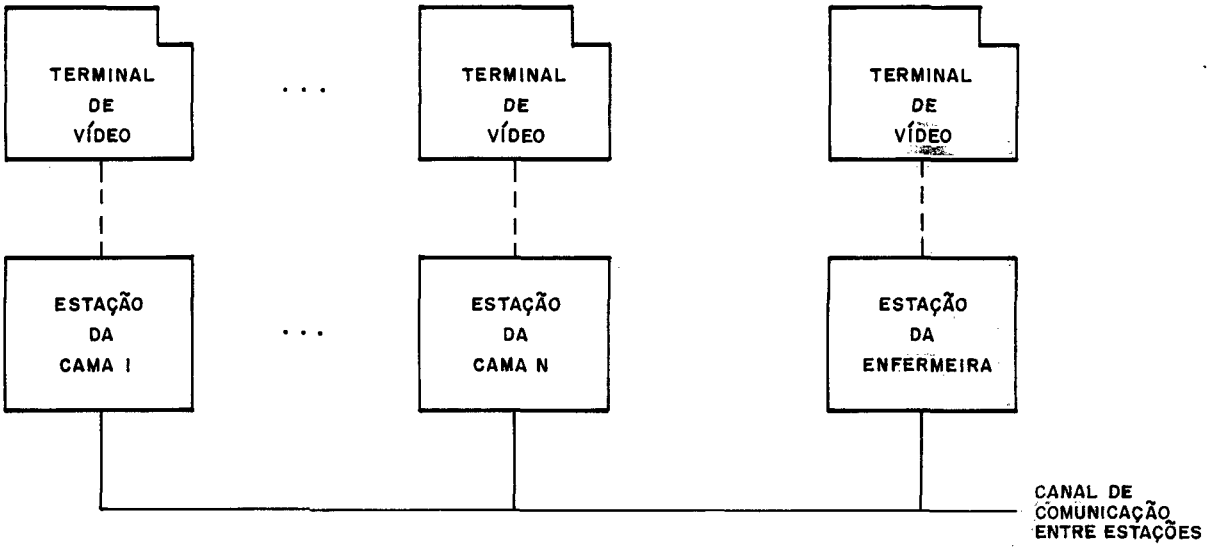

FIGURA VI.2 - SISTEMA DE CONTROLE DE PACIENTES

As estações das camas são todas iguais, contendo um terminal de video onde são mostrados os valores dos fatores que estão sendo controlados, e através do qual são introduzidos os limites de variação de segurança. A estação da enfermaria emite alarmes, e pode querer repetir no seu terminal de video os valores dos fatores que estão sendo controlados em cada paciente.

Cada estação lógica, que será carregada numa única estação física, é constituida por um conjunto de módulos ligados, que executam as funções da estação, e poderão pertencer a mais de uma configuração, e não necessariamente a uma só.

Para especificar este sistema definimos três tipos de configurações, uma correspondente à Configuração do Paciente, da qual serão criadas tantas instâncias quantas camas existem, uma Configuração da Enfermeira, e uma Configuração da Enfermaria, composta pelas instâncias criadas a partir das duas configurações anteriores.

A Configuração da Enfermeira utiliza unicamente a estação lógica da Enfermeira. A Configuração do Paciente precisa de duas estações lógicas, a dele e a da enfermeira, já que algumas das funções são executadas perto da cama e outras são executadas na estação da enfermeira, tais como acionar o alarme e mostrar os dados do paciente no video da enfermeira, quando estes forem requisitados. Estas funções precisam de módulos específicos para cada cama.

Vamos então mostrar a especificação da configuração do sistema usando nossa linguagem de configuração, sem nos preocupar em explicar os detalhes das funções de cada módulo que estão programados em Modula-2. A descrição completa do sistema seria muito grande, e para efeitos deste trabalho achamos suficiente a abordagem escolhida.

Descrevemos a seguir, sucintamente, as funções dos tipos de módulos utilizados para formar a configuração do exemplo apresentado:

psim: este módulo simula as entradas dos fatores do paciente a serem controlados e importa a interface de pmonit.

pmonit: este módulo compara as leituras recebidas com os limites de variação e provoca um alarme quando o valor estiver fora do limite. Armazena todas as informações

do paciente para poder fornecê-las quando forem solicitadas. Ele exporta uma interface para psim e importa as interfaces de palarm, de psel e de pdisp.

pdisp: este módulo recebe informações de um cliente para formatá-las e encaminhá-las para serem exibidas. Ele importa a interface de kwind e exporta uma interface para o cliente.

pcom: este módulo permite que sejam introduzidas novas informações do paciente através do console da cama do paciente. Ele importa as interfaces de pmonit e de zterm.

palarm: este módulo aceita as mensagens de alarme enviadas a partir das camas e as formata para que sejam exibidas. Ele exporta uma interface para pmonit e importa a interface de kwind.

kwind: este módulo formata a tela do console. Ele exporta uma interface para pdisp e zlogw e importa a interface de zterm.

zlogw: este módulo formata os erros para serem exibidos na tela. Ele importa a interface de kwind.

psel: este módulo recebe informações das camas e as armazena para tê-las disponíveis para pedidos da enfermeira. Ele exporta uma interface para nursecom e pmonit e importa a interface de pdisp.

nursecom: este módulo executa as funções de interpretação dos comandos que permitem à enfermeira escolher uma cama para monitorar. Ele importa as interfaces de zterm e de psel.

zterm: este módulo corresponde ao gerenciador de console e contém quatro tarefas internas que gerenciam respectivamente a leitura de linha, escrita de linha, a tela e o teclado.

Este exemplo se caracteriza por ter um único tipo de módulo de implementação para cada interface de módulo e por possuir dois

níveis de hierarquia de configurações. No primeiro nível são definidas as sub-configurações PatientConf e NurseConf, a serem combinadas no segundo nível para formar a configuração WardConf, na qual casam as importações de interfaces de módulos com as exportações de módulos de implementação. Como neste exemplo só tem um módulo de implementação para cada módulo de interface, usamos o mesmo nome para os dois, nas configurações definidas a seguir.

A Configuração do Paciente, considerando os módulos principais que a constituem, é ilustrada na Figura (VI.3):

```
CONFIGURATION :PatientConf;
   USE zterm,pmonit,pcom,kwind,pdisp,psim,zlogw,palarm,psel;
   STATIONS Bed,Nurse;
   IMPORT Select : psel,ConsoleN:zterm;
   CREATE
      Console:zterm ON Bed;
      Monitor:pmonit ON Bed;
      Command:pcom ON Bed;
      ErrorWindow, Window:kwind ON Bed;
      Display:pdisp ON Bed;
      Scanner:psim ON Bed;
      ErrorFormat:zlogw ON Bed;
      AlarmN:palarm ON Nurse;
      WindowN:kwind ON Nurse;

   LINK
      WindowN WITH ConsoleN;
      AlarmN WITH WindowN;
      Monitor WITH Select,AlarmN,Display;
      Command WITH Monitor,Console;
      ErrorWindow,Window WITH Console;
      Display WITH Window;
      Scanner WITH Monitor;
      ErrorFormat WITH ErrorWindow;
END PatientConf.
```

FIGURA VI.3 - DECLARAÇÃO DA SUB-CONFIGURAÇÃO DO PACIENTE

As ligações dentro desta configuração ligam instâncias de módulos locais entre si, e instâncias de módulos de implementação desta configuração com interfaces de módulos de outras configurações, explicitadas na lista de importações. Os nomes das interfaces dos módulos importadas são locais a este tipo de configuração, e serão substituidos por instâncias de módulos reais na hora de definir as ligações das configurações, como será visto mais adiante.

Convém destacar que, já que as estações lógicas Bed e Nurse serão mapeadas em estações físicas diferentes, a terceira declaração de LINK estabelece uma ligação remota, já que o módulo AlarmN pertence à estação Nurse, enquanto Monitor pertence à

estação Bed. Podemos observar também, que a ligação de Monitor com Select é potencialmente remota, mas só poderemos confirmá-lo quando for ligada esta sub-configuração com outra no próximo nível da hierarquia. A primeira declaração de ligação é local porque os dois módulos pertencem à mesma estação Nurse.

A Configuração da Enfermeira, considerando os módulos principais que a constituem, é ilustrada na Figura (VI.4):

```
CONFIGURATION: NurseConf;
   USE nursecom,pdisp,psel,zterm,zlogw,kwind;
   STATIONS Nurse;
   EXPORT Selector:psel,ConsoleNurse:zterm;
   CREATE
      Command:nursecom ON Nurse;
      Display:pdisp ON Nurse;
      Selector:psel ON Nurse;
      ConsoleNurse:zterm ON Nurse;
      ErrorFormat:zlogw ON Nurse;
      ErrorWindow,Window:kwind ON Nurse;

   LINK
      Window,ErrorWindow WITH ConsoleNurse;
      ErrorFormat WITH ErrorWindow;
      Display WITH Window;
      Command WITH Selector,ConsoleNurse;
      Selector WITH Display;
END NurseConf.
```

FIGURA VI.4 - DECLARAÇÃO DA SUB-CONFIGURAÇÃO DA ENFERMEIRA

Passaremos agora a definir a Configuração da Enfermaria, que é composta por instâncias das Configurações já declaradas e é ilustrada na Figura (VI.5).

```
CONFIGURATION: WardConf;
   USE NurseConf,PatientConf;
   STATIONS Bed1,Bed2,Bed3,Bed4,Bed5,Nurse;
   CREATE
      Maria:NurseConf ON Nurse;
      Pat1:PatientConf ON Bed1,Nurse;
      Pat2:PatientConf ON Bed2,Nurse;
      Pat3:PatientConf ON Bed3,Nurse;
      Pat4:PatientConf ON Bed4,Nurse;
      Pat5:PatientConf ON Bed5,Nurse;

   LINK
      Pat1,Pat2,Pat3,Pat4,Pat5
      WITH Maria.Selector,Maria.ConsoleNurse;
END WardConf.
```

FIGURA VI.5 - DECLARAÇÃO DA CONFIGURAÇÃO DA ENFERMARIA

O sistema aqui descrito é composto por uma enfermeira controlando cinco camas de pacientes. Podemos ressaltar que a

configuração de mais alto nível não exporta nem importa nada. É nesta configuração que são feitas as ligações entre as Configurações de Pacientes e a Configuração da Enfermeira através da associação entre as interfaces dos módulos importadas pelos pacientes e as instâncias dos módulos de implementação exportadas pela enfermeira.

Para completar este exemplo, podemos explicitar o mapeamento da configuração lógica na configuração física através de:

```
LOAD Enfermaria:WardConf ON Station1,Station2,Station3,Station4,
                            Station5,Station6.
```

VI.2.2 EXEMPLO DE PROPAGAÇÃO DE IMPORTAÇÕES E EXPORTAÇÕES

Este exemplo tem como objetivo mostrar a propagação de importações de interfaces de módulos e de exportações de módulos de implementação. Para isto foi estendido o exemplo da seção anterior, acrescentando dois tipos de sub-configurações no primeiro nível de hierarquia, chamados ControlConf e DoctorConf. Foi acrescentado também um terceiro nível de hierarquia de tipo de configuração chamado FloorConf, formado a partir das sub-configurações WardConf, ControlConf e DoctorConf.

A sub-configuração ControlConf tem como função fazer algum cálculo estatístico sobre os dados dos pacientes, sendo que para isto obtém informações através da NurseConf. A sub-configuração DoctorConf desempenha o papel do doutor que será consultado em determinadas situações pela sub-configuração NurseConf. A configuração de terceiro nível FloorConf representa um andar de um hospital, no qual estão alocados um doutor, uma enfermeira e cinco pacientes. É neste último nível que vão casar todas as importações de interfaces de módulos com as exportações de módulos de implementação.

Para melhor entendimento do exemplo apresentado ilustramos resumidamente a sua estrutura através da Figura (VI.6).

1o. nível da hierarquia

Tipos de sub-configurações

PatientConf
    importa de NurseConf

NurseConf
    importa de DoctorConf
    exporta para PatientConf e ControlConf

ControlConf
    importa de NurseConf

DoctorConf
    exporta para NurseConf

2o. nível da hierarquia

Tipo de sub-configuração

WardConf
    importa de DoctorConf
    exporta para ControlConf
  1 instância do tipo NurseConf
  5 instâncias do tipo PatientConf

3o. nível da hierarquia

Tipo de configuração

FloorConf
    1 instância do tipo WardConf
    1 instância do tipo ControlConf
    1 instância do tipo DoctorConf

FIGURA VI.6 - ESQUEMA DA CONFIGURAÇÃO DO EXEMPLO DE FloorConf COM CONTROLE ESTATÍSTICO

Queremos destacar que na sub-configuraçãp WardConf do segundo nível da hierarquia, da Figura (VI.6), casam as importações de interfaces de módulos de PatientConf com as exportações de módulos de implementação de NurseConf. É neste nível que mostramos a propagação tanto da importação declarada em DoctorConf quanto da exportação declarada em NurseConf, que só serão casadas no terceiro nível de hierarquia, na configuração FloorConf.

Apresentaremos a seguir, com mais detalhes, a definição das sub-configurações e da configuração mencionadas.

A definição da sub-configuração PatientConf permanece inalterada em relação ao exemplo anterior e é ilustrada na Figura (VI.3).

Para este exemplo é necessário acrescentar à definição anterior da sub-configuração NurseConf a importação da interface

do módulo do tipo consDoctor, contido na sub-configuração DoctorConf, e a exportação do módulo de implementação correspondente à interface do módulo importada na sub-configuração ControleConf, do tipo xtipo.

```
CONFIGURATION: NurseConf;
   USE nursecom,pdisp,psel,zterm,zlogw,kwind,xtipo,
              ytipo,consDoctor;
   STATIONS Nurse;
   IMPORT ConsultDoctor:consDoctor;
   EXPORT Selector:psel,ConsoleNurse:zterm,X:xtipo;
   CREATE
      Command:nursecom ON Nurse;
      Display:pdisp ON Nurse;
      Selector:psel ON Nurse;
      ConsoleNurse:zterm ON Nurse;
      X:xtipo ON Nurse;
      Y:ytipo ON Nurse;
      ErrorFormat:zlogw ON Nurse;
      ErrorWindow,Window:kwind ON Nurse;

   LINK
      Window,ErrorWindow WITH ConsoleNurse;
      ErrorFormat WITH ErrorWindow;
      Display WITH Window;
      Command WITH Selector,ConsoleNurse;
      Selector WITH Display;
      X WITH Selector;
      Y WITH ConsultDoctor;
      X WITH Y;
END NurseConf.
```

FIGURA VI.7 - DECLARAÇÃO DA SUB-CONFIGURAÇÃO NurseConf

Os módulos X e Y foram acrescentados para fazer consultas ao doutor em relação às informações que precisam ser utilizadas para os cálculos estatísticos, a partir dos dados recolhidos dos pacientes.

A definição da sub-configuração de ControlConf é ilustrada pela Figura (VI.8).

```
CONFIGURATION: ControlConf;
   USE wtipo,ztipo,xtipo;
   STATIONS Nurse;
   IMPORT X:xtipo;
   CREATE
      W:wtipo ON Nurse;
      Z:ztipo ON Nurse;

   LINK
      W WITH Z,X;
END ControlConf.
```

FIGURA VI.8 - DECLARAÇÃO DA SUB-CONFIGURAÇÃO ControlConf

Esta sub-configuração faz uns cálculos estatísticos utilizando os módulos W e Z, e importando, através de X, dados

fornecidos pela enfermeira. Esta importação será casada num nível superior de hierarquia.

A última sub-configuração neste primeiro nível de hierarquia é a DoctorConf, cuja definição é ilustrada pela Figura (VI.9).

```
CONFIGURATION: DoctorConf;
   USE consDoctor;
   STATIONS Doctor;
   EXPORT CD:consDoctor;
   CREATE
      CD:consDoctor ON Doctor;
END DoctorConf.
```

FIGURA VI.9 - DECLARAÇÃO DA SUB-CONFIGURAÇÃO DoctorConf

Esta sub-configuração consiste num único módulo de implementação cuja interface será importada pela enfermeira e portanto não contém nenhuma ligação.

No segundo nível de hierarquia, este exemplo define uma sub-configuração chamada WardConf, que consiste de uma instância de enfermeira e cinco instâncias de pacientes, analogamente ao exemplo anterior.

Por causa das modificações acrescentadas à sub-configuração NurseConf, a definição ilustrada pela Figura (VI.10) mostra algumas alterações em relação à versão anterior da Figura (VI.5).

```
CONFIGURATION: WardConf;
   USE NurseConf,PatientConf,xtipo,consDoctor;
   STATIONS Bed1,Bed2,Bed3,Bed4,Bed5,Nurse;
   IMPORT ConsultDoctor:consDoctor;
   EXPORT X:xtipo;
   CREATE
      Maria:NurseConf ON Nurse;
      Pat1:PatientConf ON Bed1,Nurse;
      Pat2:PatientConf ON Bed2,Nurse;
      Pat3:PatientConf ON Bed3,Nurse;
      Pat4:PatientConf ON Bed4,Nurse;
      Pat5:PatientConf ON Bed5,Nurse;

   LINK
      Pat1,Pat2,Pat3,Pat4,Pat5
      WITH Maria.Selector,Maria.ConsoleNurse;
END WardConf.
```

FIGURA VI.10 - DECLARAÇÃO DA SUB-CONFIGURAÇÃO WardConf

Como já mencionamos anteriormente, neste segundo nível da hierarquia aparecem as propagações, por um lado, da importação da interface do módulo ConsultDoctor, utilizada pela enfermeira, e,

por outro, da exportação do módulo de implementação X contido na enfermeira.

Para finalizar este exemplo, mostraremos a definição do terceiro e último nível de configuração chamado FloorConf, que consiste de uma instância de cada uma das três sub-configurações WardConf, ControlConf e DoctorConf, ilustrada pela Figura (VI.11).

```
CONFIGURATION: FloorConf;
   USE WardConf,ControlConf,DoctorConf;
   STATIONS Bed1,Bed2,Bed3,Bed4,Bed5,Nurse,Doctor;
   CREATE
      Enfermaria:WardConf ON Bed1,Bed2,Bed3,Bed4,Bed5,Nurse;
      ControleEstatistico:ControlConf ON Nurse;
      José:DoctorConf ON Doctor;

   LINK
      ControleEstatistico WITH Enfermaria.X;
      Enfermaria WITH José.CD;
END FloorConf.
```

FIGURA VI.11 - DECLARAÇÃO DA CONFIGURAÇÃO FloorConf

Podemos notar que neste último nível casam a importação e a exportação propagadas, que tinham ficado pendentes. Queremos salientar também, que a importação de ConsultDoctor é feita através da instância Enfermaria da sub-configuração do tipo WardConf, mas ela só é visível para a instância Maria da sub-configuração do tipo NurseConf, e não é visível para nenhuma das instâncias das sub-configurações do tipo PatientConf contidas nesta instância Enfermaria.

VI.2.3 EXEMPLO DE USO DE INSTÂNCIAS DE TIPOS DE MÓDULOS E SUB-CONFIGURAÇÕES IGUAIS

No exemplo anterior vimos o uso de várias instâncias de uma mesma sub-configuração (PatientConf), que em particular continha declarações de importação. Neste exemplo queremos mostrar o uso de várias instâncias de uma mesma sub-configuração, tanto no caso de sub-configurações que importam, quanto das que exportam, já que estas últimas apresentam alguns problemas adicionais.

Neste caso desenvolveremos o exemplo do andar do hospital descrito antes, mas com a ressalva de que tiramos a sub-configuração do cálculo estatístico do tipo ControlConf por razões que explicaremos mais adiante. O exemplo vai conter então

o mesmo doutor, mas uma enfermaria diferente, com duas enfermeiras do mesmo tipo, e aparecerá somente a propagação da importação já mostrada no exemplo anterior.

A estrutura do exemplo pode ser ilustrada resumidamente pela Figura (VI.12).

1o. nível da hierarquia

Tipos de sub-configurações

PatientConf
   importa de NurseConf

NurseConf
   importa de DoctorConf
   exporta para PatientConf

DoctorConf
   exporta para NurseConf

2o. nível da hierarquia

Tipo de sub-configuração

WardConfT
   importa de DoctorConf
 2 instâncias do tipo NurseConf
 5 instâncias do tipo PatientConf

3o. nível da hierarquia

Tipo de configuração

FloorConf1
   1 instância do tipo WardConfT
   1 instância do tipo DoctorConf

FIGURA VI.12 - ESQUEMA DA CONFIGURACAO DO EXEMPLO DE FloorConf1

Neste caso, tendo duas enfermeiras iguais para atender aos cinco pacientes, precisamos ligar explicitamente cada paciente a uma determinada enfermeira. Isto é consequência do fato que, quando o paciente faz a chamada remota do procedimento contido na interface exportada pela enfermeira, não é possível precisar a que instância corresponde essa chamada. Esta correspondência é feita de forma fixa na declaração **LINK**. Por causa disto, é que tivemos que retirar a sub-configuração ControlConf, que importava uma interface da sub-configuração NurseConf. Tendo duas enfermeiras não poderíamos diferenciar, na hora de ControlConf fazer a chamada a NurseConf, qual delas estaria chamando, mesmo fazendo as duas ligações de ControlConf com elas.

As sub-configurações do primeiro nível da hierarquia são então as três mencionadas na Figura (VI.12), sendo que a única diferença em relação ao exemplo anterior é que a sub-configuração do tipo NurseConf agora não exporta mais para ControlConf, que agora foi retirado. As declarações de PatientConf e DoctorConf, correspondem às Figuras (VI.3 e VI.9). A declaração de NurseConf será ilustrada pela Figura (VI.13).

```
CONFIGURATION: NurseConf;
   USE nursecom,pdisp,psel,zterm,zlogw,kwind,
            ytipo,consDoctor;
   STATIONS Nurse;
   IMPORT ConsultDoctor:consDoctor;
   EXPORT Selector:psel,ConsoleNurse:zterm;
   CREATE
      Command:nursecom ON Nurse;
      Display:pdisp ON Nurse;
      Selector:psel ON Nurse;
      ConsoleNurse:zterm ON Nurse;
      Y:ytipo ON Nurse;
      ErrorFormar:zlogw ON Nurse;
      ErrorWindow,Window:kwind ON Nurse;

   LINK
      Window,ErrorWindow WITH ConsoleNurse;
      ErrorFormat WITH ErrorWindow;
      Display WITH Window;
      Command WITH Selector,ConsoleNurse;
      Selector WITH Display;
      Y WITH ConsultDoctor;
END NurseConf.
```

FIGURA VI.13 - DECLARAÇÃO DA SUB-CONFIGURAÇÃO NurseConf

Foram retirados da sub-configuração NurseConf anterior, ilustrada na Figura (VI.7), a declaração e uso do módulo X de tipo xtipo que tínhamos colocado para interagir com a sub-configuração ControlConf.

No segundo nível criamos uma nova sub-configuração do tipo WardConfT, a partir do primeiro exemplo (seção VI.2.1), no qual acrescentamos uma outra instância da sub-configuração NurseConf. A nova sub-configuração WardConfT estará formada então por duas instâncias de enfermeiras e cinco instâncias de pacientes, e a sua definição é ilustrada na Figura (VI.14).

```
CONFIGURATION: WardConfT;
   USE NurseConf,PatientConf,ConsultDoctor;
   STATIONS Bed1,Bed2,Bed3,Bed4,Bed5,Nurse1,Nurse2;
   IMPORT ConsultDoctor:consDoctor;
   CREATE
      Maria:NurseConf ON Nurse1;
      Ana:NurseConf ON Nurse2;
      Pat1:PatientConf ON Bed1,Nurse1;
      Pat2:PatientConf ON Bed2,Nurse1;
      Pat3:PatientConf ON Bed3,Nurse1;
      Pat4:PatientConf On Bed4,Nurse2,
      Pat5:PatientConf ON Bed5,Nurse2;

   LINK
      Pat1,Pat2,Pat3 WITH Maria.Selector,Maria.ConsoleNurse;
      Pat4,Pat5 WITH Ana.Selector,Ana.ConsoleNurse;
END WardConfT.
```

FIGURA VI.14 - DECLARAÇÃO DA SUB-CONFIGURAÇÃO WardConfT

Queremos ressaltar, que nesta declaração de WardConfT temos ligado os três primeiros pacientes com a enfermeira Maria, e os outros dois com a enfermeira Ana. As duas enfermeiras foram associadas a estações lógicas diferentes mas poderiam também estar associadas à mesma estação lógica, e neste caso a declaração teria sido:

```
CREATE
   Maria,Ana:NurseConf ON Nurse;
```

e os pacientes teriam sido declarados todos na mesma estação Nurse.

No terceiro nível temos então a definição da configuração do tipo FloorConf1 ilustrada pela Figura (VI.15).

```
CONFIGURATION: FloorConf1;
   USE WardConfT,DoctorConf;
   STATIONS Bed1,Bed2,Bed3,Bed4,Bed5,Nurse1,Nurse2,Doctor;
   CREATE
      EnfermariaT:WardConfT ON Bed1,Bed2,Bed3,Bed4,Bed5,Nurse1,
                               Nurse2;
      José:DoctorConf ON Doctor;

   LINK
      EnfermariaT WITH José.CD;
END FloorConf1.
```

FIGURA VI.15 - DECLARAÇÃO DA CONFIGURAÇÃO FloorConf1

Podemos notar que neste último nível da hierarquia se casa a importação propagada pela sub-configuração WardConfT da interface de ConsultDoctor com a exportação da instância do módulo de implementação do tipo consDoctor correspondente, propagada pela configuração DoctorConf. Mas o fato mais importante a ser

ressaltado é que a ligação declarada implica em duas ligações implícitas que o interpretador da linguagem de configuração precisa identificar, já que na sub-configuração WardConfT existem duas instâncias, Maria e Ana, da sub-configuração NurseConf que importam a interface de ConsultDoctor.

Este exemplo podia ter sido estruturado, separando a sub-configuração WardConfT em duas, WardConf1 e WardConf2, contendo, respectivamente, a enfermeira Maria com seus três pacientes, e a enfermeira Ana com seus dois pacientes. Esta estrutura não teria introduzido nenhuma diferença conceitual para a configuração final; a única mudança seria em relação às duas ligações explícitas

```
Enfermaria1 WITH José.CD;
Enfermaria2 WITH José.CD;
```

supondo que Enfermaria1 fosse do tipo WardConf1 e Enfermaria2 do tipo WardConf2.

Voltamos a ressaltar aqui, que quando existem mais de uma instância de uma sub-configuração, sejam elas do mesmo tipo ou de tipos diferentes, não podemos diferenciá-las no caso delas exportarem alguma interface que seja importada por outra sub-configuração. Esta limitação é consequência da propriedade de transparência na chamada remota de procedimento, que quisemos implementar para estender Modula-2 para o seu uso em sistemas distribuídos. Queremos deixar claro que uma sub-configuração no primeiro nível da hierarquia está constituída por módulos escritos em Modula-2, e, quando falamos de instâncias de sub-configurações de tipos diferentes, isto corresponde a sub-configurações contendo módulos de implementação de tipos diferentes, mas com a mesma interface. Isto ocorre tanto em relação a módulos locais à sub-configuração quanto em relação a módulos que são visíveis para as outras sub-configurações, por terem sido exportados. Trataremos deste caso no próximo exemplo.

VI.2.4 EXEMPLO DE USO DE INSTÂNCIAS DE TIPOS DIFERENTES DE MÓDULOS E SUB-CONFIGURAÇÕES

Até agora temos tratado exemplos, nos quais para cada

interface de módulo existia somente um tipo de módulo de implementação, mesmo com várias instâncias iguais, como no caso anterior. Queremos agora apresentar um exemplo no qual, para uma mesma interface de módulo, existem diferentes tipos de módulos de implementação programados em Modula-2. Para isto escolhemos o caso de ter dentro da sub-configuração do tipo NurseConf tipos de módulos de implementação diferentes para a interface psei, que em particular são exportados. Poderíamos ter escolhido um tipo de módulo local à NurseConf que não fosse exportado, mas o caso seria análogo só que mais simples.

Vamos estão definir aqui os nomes diferentes para as interfaces dos módulos e para os nomes dos módulos de implementação correspondentes. Chamaremos a interface do módulo psei, pseiint, e os dois tipos de módulos de implementação, psei1

1o. nível da hierarquia

Tipos de sub-configuração

PatientConf
   importa de NurseConf1 ou de NurseConf2

NurseConf1
   importa de DoctorConf
   exporta para PatientConf

NurseConf2
   importa de DoctorConf
   exporta para PatientConf

DoctorConf
   exporta para NurseConf1 e NurseConf2

2o.nível da hierarquia

Tipo de sub-configuração

WardConf1
   importa de DoctorConf
 1 instância do tipo NurseConf1
 3 instâncias do tipo PatientConf

WardConf2
   importa de DoctorConf
 1 instância do tipo NurseConf2
 2 instâncias do tipo PatientConf

3o. nível da hierarquia

Tipo de configuração

FloorConf2
   1 instância do tipo WardConf1
   1 instância do tipo WardConf2
   1 instância do tipo DoctorConf

FIGURA VI.16 - ESQUEMA DA CONFIGURAÇÃO DO EXEMPLO DE FloorConf2

e psel2 respectivamente. Vamos supôr também que para o módulo do tipo zterm existem dois nomes diferentes, um para a interface do módulo, chamada ztermint, e outro para o módulo de implementação, chamado zterm, mesmo tendo um único tipo de módulo de implementação.

Para desenvolver o exemplo análogo ao anterior, precisamos agora definir dois tipos diferentes de sub-configurações para as duas enfermeiras. Para facilitar a compreensão do exemplo de FloorConf2, ilustraremos resumidamente a sua estrutura com a Figura (VI.16) acima.

Com a definição de nomes diferentes para as interfaces dos módulos e para os tipos dos módulos de implementação, analisemos agora as modificações a serem introduzidas em relação às definições das sub-configurações apresentadas anteriormente.

No primeiro nível da hierarquia vejamos agora as declarações dos tipos de sub-configurações. A declaração do tipo de sub-configuração PatientConf será agora ilustrado pela Figura (VI.17)

```
CONFIGURATION :PatientConf;
   USE zterm,pmonit,pcom,kwind,pdisp,psim,zlogw,palarm,
               pselint,ztermint;
   STATIONS Bed,Nurse;
   IMPORT Select : pselint,ConsoleN:ztermint;
   CREATE
      Console:zterm ON Bed;
      Monitor:pmonit ON Bed;
      Command:pcom ON Bed;
      ErrorWindow, Window:kwind ON Bed;
      Display:pdisp ON Bed;
      Scanner:psim ON Bed;
      ErrorFormat:zlogw ON Bed;
      AlarmN:palarm ON Nurse;
      WindowN:kwind ON Nurse;

   LINK
      WindowN WITH ConsoleN;
      AlarmN WITH WindowN;
      Monitor WITH Select,AlarmN,Display;
      Command WITH Monitor,Console;
      ErrorWindow,Window WITH Console;
      Display WITH Window;
      Scanner WITH Monitor;
      ErrorFormat WITH ErrorWindow;
END PatientConf.
```

FIGURA VI.17 - DECLARAÇÃO DA SUB-CONFIGURAÇÃO PatientConf

Podemos destacar que as únicas diferenças aparecem nas

declarações **USE** e **IMPORT**, onde foram acrescentados os nomes das interfaces de módulos importadas, pselint e ztermint. Na declaração **CREATE** é criada uma instância do tipo zterm, que continua igual à anterior, já que estamos supondo que existe um único tipo de módulo correspondente à interface ztermint.

Neste exemplo precisamos definir duas sub-configurações de tipos diferentes para NurseConf, que serão chamadas respectivamente NurseConf1 e NurseConf2, contendo tipos de módulos de implementação diferentes, psel1 e psel2, para o módulo psel usado anteriormente.

A sub-configuração do tipo NurseConf1 é ilustrada pela Figura (VI.18).

```
CONFIGURATION: NurseConf1;
   USE nursecom,pdisp,psel1,zterm,zlogw,kwind,
            ytipo,consDoctor;
   STATIONS Nurse;
   IMPORT ConsultDoctor:consDoctor;
   EXPORT Selector:psel1,ConsoleNurse:zterm;
   CREATE
      Command:nursecom ON Nurse;
      Display:pdisp ON Nurse;
      Selector:psel1 ON Nurse;
      ConsoleNurse:zterm ON Nurse;
      Y:ytipo ON Nurse;
      ErrorFormat:zlogw ON Nurse;
      ErrorWindow,Window:kwind ON Nurse;

   LINK
      Window,ErrorWindow WITH ConsoleNurse;
      ErrorFormat WITH ErrorWindow;
      Display WITH Window;
      Command WITH Selector,ConsoleNurse;
      Selector WITH Display;
      Y WITH ConsultDoctor;
END NurseConf.
```

FIGURA VI.18 - DECLARAÇÃO DA SUB-CONFIGURAÇÃO NurseConf1

A única modificação introduzida na Figura (VI.18) em relação á Figura (VI.13) é a substituição de psel por psel1.

Analogamente a sub-configuração do tipo NurseConf2 é ilustrada pela Figura (VI.19) com a única modificação causada pela substituição de psel por psel2.

```
CONFIGURATION: NurseConf2;
   USE nursecom,pdisp,psel2,zterm,zlogw,kwind,
            ytipo,consDoctor;
   STATIONS Nurse;
   IMPORT ConsultDoctor:consDoctor;
   EXPORT Selector:psel2,ConsoleNurse:zterm;
   CREATE
      Command:nursecom ON Nurse;
      Display:pdisp ON Nurse;
      Selector:psel2 ON Nurse;
      ConsoleNurse:zterm ON Nurse;
      Y:ytipo ON Nurse;
      ErrorFormat:zlogw ON Nurse;
      ErrorWindow,Window:kwind ON Nurse;

   LINK
      Window,ErrorWindow WITH ConsoleNurse;
      ErrorFormat WITH ErrorWindow;
      Display WITH Window;
      Command WITH Selector,ConsoleNurse;
      Selector WITH Display;
      Y WITH ConsultDoctor;
END NurseConf.
```

FIGURA VI.19 - DECLARAÇÃO DA SUB-CONFIGURAÇÃO NurseConf2

No segundo nível da hierarquia são definidas duas sub-configurações do tipo WardConf1 e WardConf2. A sub-configuração do tipo WardConf1 é definida pela Figura (VI.20) seguinte:

```
CONFIGURATION: WardConf1;
   USE PatientConf,NurseConf1;
   STATIONS Bed1,Bed2,Bed3,Nurse1;
   IMPORT ConsultDoctor:consDoctor;
   CREATE
      Maria:NurseConf1 ON Nurse1;
      Pat1:PatientConf ON Bed1,Nurse1;
      Pat2:PatientConf ON Bed2,Nurse1;
      Pat3:PatientConf ON Bed3,Nurse1;

   LINK
      Pat1,Pat2,Pat3 WITH Maria.Selector,Maria.ConsoleNurse;
END WardConf1.
```

FIGURA VI.20 - DECLARAÇÃO DA SUB-CONFIGURAÇÃO WardConf1

Esta sub-configuração está composta por uma instância da sub-configuração NurseConf1 e três instâncias da sub-configuração PatientConf. É neste nível que na declaração do LINK são ligadas de forma implícita as interfaces pselint, chamadas localmente de Select, dos três pacientes com o módulo de implementação de tipo psel1 de NurseConf1, chamado localmente de Selector.

Analogamente a sub-configuração do tipo WardConf2 é definida pela Figura (VI.21) seguinte:

```
CONFIGURATION: WardConf2;
   USE PatientConf,NurseConf2;
   STATIONS Bed4,Bed5,Nurse2;
   IMPORT ConsultDoctor:consDoctor;
   CREATE
      Ana:NurseConf2 ON Nurse2;
      Pat4:PatientConf ON Bed4,Nurse2;
      Pat5:PatientConf ON Bed5,Nurse2;

   LINK
      Pat4,Pat5 WITH Ana.Selector,Ana.ConsolNurse;
END WardConf2.
```

FIGURA VI.21 - DECLARAÇÃO DA SUB-CONFIGURAÇÃO WardConf2

Nesta sub-configuração, que contém uma instância da sub-configuração NurseConf2 e duas instâncias de PatientConf, são feitas implicitamente as ligações entre as interfaces pselint, chamadas localmente de Select, dos pacientes com o módulo de implementação de tipo psel2 de NurseConf2, chamado localmente de Selector.

No terceiro nível definimos a configuração do tipo FloorConf2 constituida por uma instância de cada um dos três tipos de sub-configurações WardConf1, WardConf2 e DoctorConf, ilustrada pela Figura (VI.22).

```
CONFIGURATION: FloorConf2;
   USE WardConf1,WardConf2,DoctorConf;
   STATIONS Bed1,Bed2,Bed3,Bed4,Bed5,Nurse1,Nurse2,Doctor;
   CREATE
      Enfermaria1:WardConf1 ON Bed1,Bed2,Bed3,Nurse1;
      Enfermaria2:WardConf2 ON Bed4,Bed5,Nurse2;
      José:DoctorConf ON Doctor;
    LINK
      Enfermaria1,Enfermaria2 WITH José.CD;
END FloorConf2.
```

FIGURA VI.22 - DECLARAÇÃO DA SUB-CONFIGURAÇÃO FloorConf2

Os comentários feitos no exemplo anterior são válidos aqui também. Poderíamos ter definido uma única sub-configuração do tipo WardConfT, contendo uma instância de cada tipo NurseConf1 e NurseConf2, e ligando os respectivos pacientes a uma e outra enfermeiras. Essa modificação não teria causado, neste nível, nenhuma mudança conceitual. Teria aparecido uma única ligação neste terceiro nível, que, de forma implícita, seria equivalente as duas ligações explícitas da Figura (VI.22).

O mais importante a ser destacado neste exemplo é a forma simples na qual é feita a ligação entre a interface do módulo e a

implementação do módulo desejado. Cada sub-configuração é programada sem precisar se preocupar com esta associação, que será feita na declaração **LINK**, quando forem combinadas as sub-configurações que contêm as respectivas importações e exportações. O programador da configuração só precisa estar ciente dos nomes das interfaces dos módulos, e dos diferentes tipos de módulos de implementação correspondentes às interfaces. É importante salientar aqui a separação de funções entre a criação de instâncias de módulos e sub-configurações, e as suas ligações. Isto facilita, na reconfiguração, a possibilidade de modificar ligações, já que para isto será necessário somente destruir as ligações existentes e fazer ligações novas, nos casos que não sejam modificados os módulos de implementação já existentes. Também no caso de modificar os módulos de implementação, esta separação facilita as mudanças a serem introduzidas, que serão minimizadas desta forma.

VI.2.5 EXEMPLO DE ADMINISTRADOR DE RECURSOS

Vimos que em todos os exemplos anteriores precisamos fazer ligações explícitas e fixas entre os pacientes e as enfermeiras. Como essa forma de ligação nos pareceu rígida demais e pouco eficiente, já que podem aparecer situações em que uma enfermeira esteja ociosa, pensamos em outro tipo de solução mais flexível e eficiente. A solução encontrada, além de resolver o problema citado, pode ser estendida para ser utilizada em outras situações bem mais gerais. Em lugar de fazer ligações fixas entre pacientes e enfermeiras, colocaremos no meio entre os dois tipos de sub-configurações, uma sub-configuração chamada administrador de enfermeiras, cuja função será precisamente a de fazer essas interligações de forma dinâmica. Esta solução, inspirada em parte na proposta do administrador apresentado por GENTLEMAN em [58] e já mencionada no Capítulo III, pode ser estendida para ser um gerenciador de recursos a ser utilizado em todos os casos que um sistema tenha recursos a serem compartilhados por vários componentes de forma dinâmica.

Para o nosso exemplo, o administrador de enfermeiras recebe chamadas dos pacientes, que são análogas às que os pacientes faziam para as enfermeiras. Ele enfilera estas chamadas para

serem depois encaminhadas para as enfermeiras. Estas por sua vez, chamam o administrador para perguntar se tem algum paciente para ser atendido através de uma RPC. Se tiver pedidos de pacientes na fila do administrador, ele entregará a o primeiro da fila para a enfermeira que o chamou. Desta forma a enfermeira atenderá o paciente escalonado, e depois de obter os resultados que precisam ser encaminhados para o paciente que ficou bloqueado esperando a resposta, ela chamará novamente o administrador através de outra RPC, para entregar os resultados a serem passados para o paciente. O administrador estará então agora em condições de responder à RPC feita pelo paciente. Desta forma, enquanto a enfermeira processa os dados do paciente, o administrador fica livre para receber pedidos de outros pacientes e chamadas da outra enfermeira. Caso não haja pedidos na fila do administrador, a enfermeira ficará ociosa, consultando de tempos em tempos o administrador até obter algum pedido de um paciente para ser atendido.

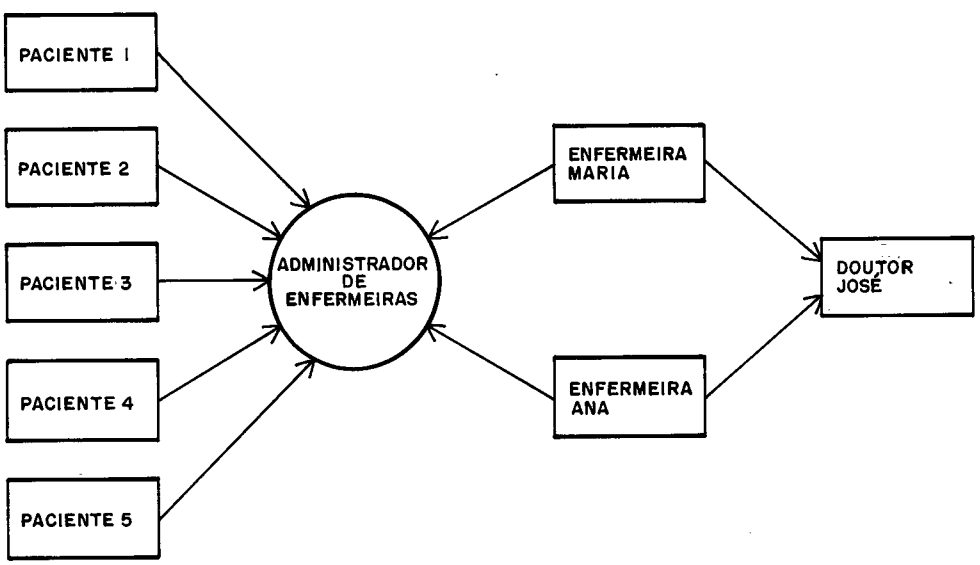

FIGURA VI.23 - EXEMPLO DO USO DO ADMINISTRADOR

O administrador de enfermeiras exportará então dois tipos de interfaces, uma para os pacientes e outra para as enfermeiras. Serão feitas portanto ligações explícitas de cada paciente e de cada enfermeira ao administrador. Desta forma as enfermeiras

serão alocadas de forma dinâmica aos pacientes e estes serão atendidos de forma mais eficiente pela primeira enfermeira que estiver disponível.

Para manter a analogia com o exemplo anterior, permitimos que as enfermeiras, para situações particulares possam consultar o doutor através de chamadas às funções exportadas por ele.

Apresentaremos neste caso este exemplo através de duas Figuras (VI.23 e VI.24) que ilustram a estrutura física e a estrutura hierárquica da configuração correspondente, sem entrar nos detalhes das declarações de cada uma das sub-configurações que compõem o sistema, por não considerá-los necessários.

A estrutura hierárquica da configuração deste exemplo será ilustrada pela Figura (VI.24) de forma análoga aos exemplos anteriores, considerando que a sub-configuração do administrador é do tipo AdmConf.

1o.nível da hierarquia

Tipos de sub-configurações

PatientConf
   importa de AdmConf

NurseConf
   importa de AdmConf e de DoctorConf

AdmConf
   exporta para PatientConf e NurseConf

DoctorConf
   exporta para NurseConf

2o. nível da hierarquia

Tipo de sub-configuração

WardConf3
   importa de DoctorConf
 5 instâncias do tipo PatientConf
 1 instância do tipo AdmConf
 2 instâncias do tipo NurseConf

3o. nível da hierarquia

Tipo de sub-configuração

FloorConf3
   1 instância do tipo WardConf3
   1 instância do tipo DoctorConf

FIGURA VI.24 - ESQUEMA DA CONFIGURAÇÃO DO EXEMPLO DO USO DO ADMINISTRADOR

Pela forma em que foi estruturada a hierarquia de sub-configurações, podemos notar que no segundo nível aparece a propagação da importação da interface de DoctorConf que será casada somente no terceiro nível da configuração FloorConf3.

Queremos ressaltar que, através deste exemplo, quisemos mostrar o potencial da linguagem de configuração definida, que permite projetar sistemas com alocação dinâmica de recursos. A estrutura de comunicação entre as sub-configurações do sistema é resultante do estilo imposto pelo uso da RPC para comunicação e sincronização entre os componentes do sistema

VI.3 IMPLEMENTAÇÃO DA LINGUAGEM DE CONFIGURAÇÃO

VI.3.1 INTRODUÇÃO

Como já foi mencionado na introdução e no início deste capítulo, a configuração do sistema programado em Modula-2, com as extensões já apresentadas para comunicação e sincronização, será definida através de uma linguagem específica que acabamos de descrever.

Para poder carregar o programa na arquitetura distribuída precisamos processar a sua definição. Isto será feito através de um interpretador, escrito em Modula-2, que processará a configuração numa máquina hospedeira de forma centralizada. Só após o final desta interpretação se terá condições de distribuir as partes adequadas do programa às estações físicas correspondentes. Este interpretador está sendo implementado por VETROMILLE [170], como parte da sua tese de Mestrado.

O interpretador possui duas partes. A primeira processará na máquina hospedeira as declarações da linguagem de configuração, fazendo a análise léxica e semântica. Serão geradas tabelas com as informações necessárias para definir os componentes a serem carregados em cada estação física. A segunda parte, chamada de **módulo de configuração**, já mencionado no Capítulo IV, consiste de um procedimento ligador e de tabelas de importação e exportação, distintas para cada estação física, que representam as interfaces com o PCRP e os "stubs".

Se faz necessário, portanto, manter certas informações até o término da interpretação da configuração, e para isto, são

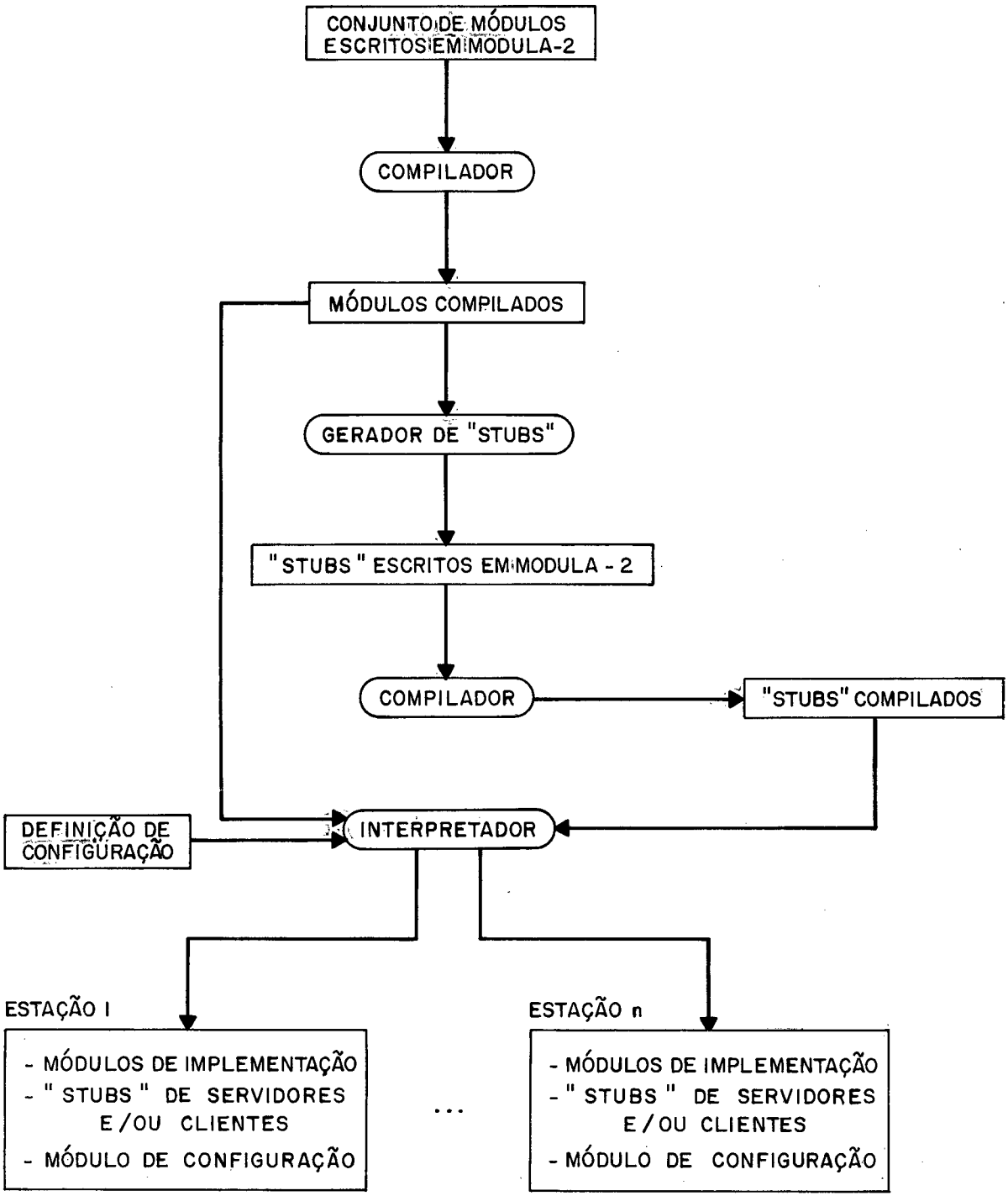

FIGURA VI.25 - ESQUEMA DAS ETAPAS QUE ANTECEDEM A EXECUÇÃO DO SISTEMA

preenchidas algumas tabelas no decorrer da análise das declarações. É necessário também montar as tabelas de importação e exportação correspondentes a cada módulo de configuração a ser carregado em cada estação física. As estruturas de dados destas tabelas serão apresentadas na próxima seção.

O sistema só se encontrará pronto para dar início a sua execução, após o final da interpretação de todas as declarações, e do carregamento nas estações físicas dos módulos de implementação, juntamente com os "stubs" de clientes e/ou servidores, e de um módulo de configuração por estação física, que será utilizado durante a execução do programa.

Vale a pena ressaltar que pode ser detectado algum erro durante a interpretação, seja na análise léxica, semântica, ou no caso de não existir "stubs" do cliente e/ou servidor. Nestes casos deverão ser tomadas medidas que não serão tratadas aqui.

Podemos mostrar, através da Figura (VI.25) acima, o esquema geral de todas as etapas que antecedem à execução de um sistema, no ambiente de programação distribuida construido a partir da linguagem Modula-2.

VI.3.2 ESTRUTURAS DE DADOS UTILIZADOS PELO INTERPRETADOR

À medida que as declarações são processadas, são montadas algumas tabelas que servirão como base para identificar os módulos de implementação correspondentes a cada estação física. As ligações remotas também são descobertas durante a interpretação e, consequentemente, obtém-se a indicação dos "stubs" dos servidores e dos clientes que devem ser carregados, e em quais estações físicas. A seguir, serão apresentadas as tabelas com um sumário das informações que elas contêm.

VI.3.2.1 TABELA DE CONFIGURAÇÕES

Cada entrada desta tabela corresponde a uma configuração ou sub-configuração definida pelo usuário através da declaração **CONFIGURATION**. Numa entrada estão contidas todas as informações correspondentes à configuração ou sub-configuração com a estrutura representada pela Figura (VI.26).

1	nome
2	tipos de módulos de implementação ou de sub-configurações e interfaces de tipos de módulos
3	estações lógicas
4	interfaces de tipos de módulos importadas
5	instâncias de módulos de implementação exportadas
6	instâncias de módulos de implementação ou sub-configuração

FIGURA VI.26 - TABELA DE CONFIGURAÇÕES

As informações da primeira parte do segundo campo são utilizadas para criar as instâncias que compõem a configuração. A segunda parte, que só existe quando houver importações, completa o contexto da configuração.

Estas informações são obtidas respectivamente no decorrer da interpretação das declarações **CONFIGURATION**, **USE**, **STATIONS**, **IMPORT**, **EXPORT** e **CREATE**.

VI.3.2.2 TABELA DE TIPOS DE CONFIGURAÇÃO

Esta tabela contém todos os tipos de configuração ou de sub-configuração criados com a declaração **CONFIGURATION**. Para cada tipo está reservada uma entrada na tabela e nela estão guardados os seguintes dados, ilustrados pela Figura (VI.27).

1	nome do tipo da configuração
2	lista de nomes das instâncias deste tipo
3	nível na hierarquia
4	apontador para a tabela de configurações

FIGURA VI.27 - TABELA DE TIPOS DE CONFIGURAÇÕES

O terceiro campo define o nível na hierarquia dentro do contexto geral da configuração do sistema ao qual este tipo pertence. Este campo é utilizado para determinar o nível mais alto, quando então será preenchida a tabela de estações lógicas descrita a seguir. Ele é utilizado também para distinguir o primeiro nível de configuração, que tem um tratamento diferente aos outros, como já foi explicado anteriormente. O campo que contém a lista dos nomes das instâncias é preenchido quando a declaração **CREATE** é interpretada.

VI.3.2.3 TABELA DAS ESTAÇÕES LÓGICAS

Cada estação lógica está associada a uma única estação física e esta relação é obtida a partir da análise da diretiva **LOAD**. Vale ressaltar que podemos encontrar configurações que utilizam mais de uma estação lógica, o que não significa que estas estações lógicas correspondam necessariamente, à mesma estação física. Para que seja possível, ao final da interpretação de todos as declarações, carregar em cada estação física o conjunto dos módulos que lhe pertencem, utiliza-se uma tabela onde cada uma de suas entradas corresponde a uma estação lógica contendo as seguintes informações, ilustrada pela Figura (VI.28).

1	nome da estação lógica
2	nome da estação física
3	apontador para o tipo i da tabela de configurações
4	lista das instâncias dos módulos de implementação
	. . . .
3	apontador para o tipo i+n da tabela de configurações
4	lista das instâncias dos módulos de implementação

FIGURA VI.28 - TABELA DE ESTAÇÕES LÓGICAS

Cada entrada desta tabela contém uma ou mais sub-configurações definidas pelo terceiro e quarto campo, que serão preenchidos somente quando for atingido o nível mais alto de configuração, para cada sub-configuração: o campo referente à estação física será preenchido depois da interpretação da diretiva **LOAD**.

VI.3.2.4 TABELAS PERTENCENTES AO MÓDULO DE CONFIGURAÇÃO

As tabelas descritas a seguir representam a interface com os "stubs", e seu uso que já foi mencionado no Capítulo IV, será mostrado nas próximas seções.

VI.3.2.4.1 TABELA DE IMPORTAÇÕES

Esta tabela, que será diferente para cada estação física, contém as informações ilustradas pela Figura (VI.29) e será identificada pelo primeiro campo.

Existirá uma entrada nesta tabela, composta pelos campos 2, 3 e 4, para cada instância de módulo que importa algum procedimento de um módulo alocado num outro nó físico, ou seja, os campos 2, 3 e 4, serão repetidos, tantas vezes quantos módulos importadores contiver esta estação física.

1	endereço da estação física
2	nome da instância do módulo que importa procedimentos
3	identificação da estação física que contém o módulo que exporta os procedimentos
4	identificação do índice da tabela de exportação que contém o endereço do despachante

FIGURA VI.29 - TABELA DE IMPORTAÇÕES

VI.3.2.4.1 TABELA DE EXPORTAÇÕES

Em cada estação física existirá esta tabela, identificada pelo primeiro campo, contendo uma entrada, composta pelos campos 2 e 3, para cada módulo exportador associando a ele o endereço do despachante, com a estrutura ilustrada pela Figura (VI.30), ou seja, os campos 2 e 3 serão repetidos tantas vezes quantos módulos exportadores contiver esta estação física.

1	endereço da estação física
2	nome da instância do módulo exportador
3	endereço do despachante

FIGURA VI.30 - TABELA DE EXPORTAÇÕES

As entradas desta tabela correspondem somente a módulos que exportam para importadores alocados em outras estações físicas.

VI.3.3 DESCRIÇÃO DO INTERPRETADOR

VI.3.3.1 PRIMEIRA PARTE DO INTERPRETADOR

Esta parte está sub-dividida em duas fases. Na primeira fase realiza-se a análise léxica e sintática das declarações, onde verifica-se a sua correta utilização, isto é, se a lei de formação da linguagem (gramática) foi respeitada pelo usuário, quando ele escreveu o programa para configurar o sistema.

A segunda fase desta primeira parte tem como objetivo interpretar a função de cada declaração, ou seja, realizar a análise semântica.

Na interpretação da declaração **CONFIGURATION**, serão alocadas uma entrada na tabela de configurações e uma na tabela de tipos de configurações. Os demais campos destas duas tabelas serão preenchidos no decorrer da interpretação das declarações **USE**, **STATIONS**, **IMPORT**, **EXPORT** e **CREATE**. Na interpretação da declaração **LINK**, é verificada a integridade da configuração, ou seja, é

conferido se os módulos que são ligados entre si estão coerentes com as exportações e importações previamente estabelecidas pelo programa do usuário e pelas definições das sub-configurações. As entradas da tabela de estações lógicas serão alocadas na interpretação da declaração **STATIONS**, pertencente à configuração de mais alto nível.

Na interpretação da diretiva **LOAD**, além de serem preenchidos os campos da estação física da tabela de estações lógicas, são determinadas as ligações locais e remotas. Uma vez obtida esta informação, é possível, percorrendo as duas primeiras tabelas, preencher a lista de instâncias dos módulos de implementação pertencentes a cada estação lógica. Neste ponto as três tabelas encontram-se totalmente preenchidas.

A partir das tabelas de configurações e de estações lógicas são montadas as tabelas de importações para cada estação física, e são alocadas as entradas das tabelas de exportação, cuja informação será preenchida posteriormente, pelos "stubs" dos servidores.

Depois de montadas estas duas últimas tabelas, não são mais necessárias as informações das primeiras três tabelas, para um ambiente de programação distribuida com configuração estática. Entretanto, se o ambiente der suporte à configuração dinâmica, estas informações deverão ser mantidas para atualizações posteriores.

Como já foi mencionado anteriormente, uma vez processada a definição da configuração é possível fazer o carregamento dos módulos de implementação, dos "stubs" de clientes e servidores e do módulo de configuração, correspondentes a cada estação física.

VI.3.3.2 SEGUNDA PARTE DO INTERPRETADOR (MÓDULO DE CONFIGURAÇÃO)

Para que o sistema possa processar corretamente uma chamada remota é necessário que o interpretador ofereça uma interface aos "stubs" do servidor e do cliente e também ao núcleo de suporte de tempo de execução (PCRP). A segunda parte do interpretador tem como função prover tais interfaces para serem utilizadas em tempo de execução. A seguir, são apresentadas as três interfaces.

1) Interface com o "stub" do cliente

Quando ocorrer uma chamada remota, a execução será desviada para o "stub" do cliente que deverá, por sua vez, requisitar um pacote ao PCRP. Ao "stub" do cliente cabe preencher alguns campos deste pacote e para isto é necessário adquirir algumas informações da tabela de importações, tais como o identificador da estação física, que contém o módulo que exporta o procedimento chamado, e o índice da tabela de exportação, que contém o endereço do despachante. Esta aquisição será realizada na inicialização dos "stubs" do cliente.

2) Interface com o "stub" do servidor

Cada "stub" do servidor, na sua fase inicial, preenche na tabela de exportações o campo do endereço do despachante correspondente a seu módulo. Quando chegar uma RPC à estação servidora, esta informação será utilizada pelo procedimento ligador para ativar o "stub" associado ao módulo que contém o procedimento chamado.

3) Interface com o PCRP

O módulo de configuração oferece ao PCRP uma interface representada pelo procedimento ligador, cujo endereço é fornecido por um dos parâmetros da rotina InitRCall do PCRP, que, como já foi mencionado no Capítulo IV, é chamada na inicialização do módulo de configuração. Quando chegar uma RPC à estação servidora o PCRP desviará a execução para este endereço.

CAPÍTULO VII

COMPARAÇÃO COM OUTRAS PROPOSTAS DE AMBIENTES DE PROGRAMAÇÃO DISTRIBUIDA

VII.1 INTRODUÇÃO

Depois de ter apresentado nos capítulos anteriores, em particular nos Capítulos IV e VI, a nossa proposta para a construção de um ambiente de programação distribuida, queremos agora compará-la com outros trabalhos encontrados na literatura. Os trabalhos escolhidos estão baseados em linguagens de programação a partir das quais pode-se construir sistemas distribuidos.

O critério de escolha das linguagens baseou-se principalmente no fato delas terem dado subsídios a nossa proposta, ou de serem as mais próximas a nosso modelo, ou ainda de serem as mais utilizadas em projetos de sistemas distribuidos e portanto existir uma experiência em relação a seu uso.

Podemos distinguir duas propriedades principais destas linguagens:

i) ter ou não linguagens separadas para especificação de configuração.

ii) dar suporte a configuração estática e dinâmica ou só estática.

Entre as linguagens que contêm funções de configuração embutidas na linguagem de programação, isto é, que não têm uma linguagem separada para especificação de configuração, escolhemos as seguintes: *MOD (COOK [42]), SR (ANDREWS e OLSSON [6]), ARGUS (LISKOV e SCHEIFLER [97]) e a proposta de adaptação de Modula-2 para sistemas distribuidos de MELLOR, DUBERY e WHITEHEAD [111].

Queremos destacar que, enquanto *MOD só da suporte a configuração estática, SR e ARGUS dão suporte à configuração dinâmica também. No caso da última proposta, a especificação da configuração não está totalmente embutida na linguagem de programação, e, apesar de ser mencionado no trabalho que é considerada a possibilidade de configuração dinâmica e reconfiguração, não é explicitado o seu processo. Podemos então

caracterizar esta última proposta como híbrida em relação às duas características apontadas anteriormente.

Entre as linguagens que apresentam uma linguagem separada para especificação de configuração, foram escolhidas as seguintes: C/MESA (MITCHELL et alii [113]), o método Mascot3 (BATE [16]) e o projeto CONIC (KRAMER e MAGEE [87], MAGEE, KRAMER e SLOMAN [106]) com a linguagem CONIC/C (DULAY et alii [52]). As duas primeiras referências especificam a configuração estática de um sistema enquanto CONIC/C dá suporte à configuração dinâmica também.

As linguagens serão apresentadas e comparadas com a nossa proposta, na ordem mencionada.

VII.2 LINGUAGEM *MOD

A linguagem *MOD foi definida por COOK em [42], derivada a partir de conceitos de Modula (WIRTH [176]) e de DP (BRINCH HANSEN [29]), com o objetivo de ser utilizada para programar sistemas e aplicações num ambiente distribuido. Os critérios mais importantes que nortearam a sua definição são a transparência e a facilidade de extensão e de adaptação.

O conceito de transparência é utilizado aqui seguindo a definição de PARNAS [125], e se refere à possibilidade de poder utilizar uma linguagem de alto nível. *MOD oferece operações de baixo nível que podem ser estendidas pelo programador, para definir abstrações de alto nível apropriadas para uma determinada arquitetura física. Os programas escritos em *MOD podem adaptar-se às mudanças de carga ou de requisitos de tempo de resposta. A linguagem, baseada em módulos, foi projetada de maneira que uma biblioteca de módulos pode ir crescendo em nível de complexidade, desde uma coleção de procedimentos até uma coleção de processos, sem modificar os programas dos usuários. A linguagem permite também ao usuário, organizar seu programa de forma a refletir a estrutura física dos processadores e das ligações da rede.

Em *MOD, foram definidos dois tipos de módulos, os **módulos do tipo rede** ("network module") e os **módulos do tipo processador** ("processor module"). Um programa é definido por um módulo do tipo rede que declara o nome e as ligações existentes entre os diferentes módulos processadores, que compõem o programa e serão

declarados a seguir. As ligações precisam ser declaradas explicitando a origem e o destino de cada uma, que correspondem a módulos do tipo processador.

A seguir são declarados os módulos de tipo processador, para os quais pode ser explicitado um número fixo de réplicas. Estes módulos foram introduzidos para permitir ao programador particionar um programa em um conjunto de processos. Todos os procedimentos e processos, que estão dentro de um módulo processador, podem ter acesso direto a variáveis compartilhadas, entanto que a comunicação entre processadores só pode ser realizada através de mensagens. Em geral, cada módulo do tipo processador é alocado num processador físico, mas é permitido alocar mais de um módulo num mesmo processador físico. Por outro lado, para aumentar a eficiência e o paralelismo é possível também alocar um mesmo módulo a mais de um processador físico.

Dentro de um módulo do tipo processador, podem ser declarados tipos de módulos, processos, procedimentos e portas. O módulo contém uma declaração de interface externa composta por declarações do tipo "define", "export" e "pervasive", e no cabeçalho do corpo do módulo pode existir a declaração "import". Vemos então que as exportações e as importações são explícitas, e as quatro declarações mencionadas satisfazem regras de escopo diferentes e implementam **interfaces de módulos bem definidas.**

Em *MOD uma rede de computadores é caracterizada por um número arbitrário de processadores, com ligações fixas entre si para transferência de mensagens. As mensagens variam num espectro que contém desde mensagens sem conteúdo, do tipo sinal ou interrupção, até as que são representadas por estruturas de dados arbitrários. Em *MOD o receptor da mensagem é chamado de **manuseador de mensagem**, e é realizado um forte controle de tipo sobre ele, o que garante a consistência do sistema. Como já foi mencionado no Capítulo III, *MOD permite unicamente que processos e portas sejam manuseadores de mensagens para comunicação entre módulos de tipo processador. Somente dentro de um processador podem ser utilizados procedimentos como manuseadores de mensagens, isto é, para comunicação intraprocessador. Utilizando estes conceitos *MOD implementa as **quatro formas de comunicação**: unidirecional síncrona e assíncrona, e bidirecional síncrona e assíncrona.

Os módulos estão compostos por procedimentos, processos e portas que são referenciados com a mesma sintaxe, de maneira que o usuário dos serviços de um módulo não precisa saber como o serviço é provido. A vantagem disto é que a implementação pode ser facilmente modificada para se adaptar a mudanças, como já foi mencionado anteriormente. A única restrição sobre estas mudanças é que procedimentos não podem ser utilizados para comunicação entre processadores.

Um processo provê execução paralela para cada manuseador de mensagens, mas requer alocação de memória para criar a sua pilha. Por outro lado, uma porta pode ser gerenciada por um único processo que permanece sempre ativo de maneira que a comunicação através da porta requeira somente a troca de contexto, que pode ser feita muito rapidamente no hardware moderno.

Queremos enfatizar que em *MOD a comunicação entre processadores é análoga, especialmente para o tipo envio/resposta, à chamada remota de procedimento, tanto através de processos quanto de portas, já que estas últimas têm trechos de programas associados.

Num dos exemplos que ilustram a definição de *MOD em [175], dentro dos módulos de tipo processador são declarados processos chamados de comunicadores, que tomam conta da comunicação entre processadores. Dentro deles estão declaradas as portas que são utilizadas para a comunicação entre processadores. O conceito de porta foi introduzido para implementar comunicação do tipo co-rotina, "rendezvous" e gerenciadores de protocolos múltiplos. No Capítulo III foi analisado em particular o uso de portas em *MOD.

Gostaríamos de destacar que esta linguagem permite fazer unicamente **configuração estática**, e que, de forma similar à proposta de extensão de Modula-2, as ligações entre módulos de tipo processador pertencem ao código de programação. Não existe uma linguagem de configuração separada, mas o carregamento na rede física é feito depois do programa ter sido compilado de forma centralizada numa máquina hospedeira, que gera código para diferentes máquinas alvo. Infelizmente não achamos maiores detalhes que nos permitam fazer uma comparação muito exaustiva.

COMPARAÇÃO DE *MOD COM A NOSSA PROPOSTA

A característica principal de *MOD é ser uma linguagem para programação distribuida, tendo como objetivo refletir a arquitetura física na qual o programa será executado. Por isto o programa define, logo no início, as ligações entre os módulos do tipo processador, que serão declarados a seguir. Estes módulos correspondem a unidades de carregamento nas estações físicas, e seus componentes podem compartilhar variáveis. Podem ser criadas várias instâncias de um mesmo tipo de módulo processador mas com o mesmo nome, e as instâncias serão distribuidas pelo índice correspondente.

Portanto a linguagem, que é utilizada tanto para a programação em pequena escala quanto para definir a configuração estática do sistema, além de não separar estas duas funções, não permite configuração dinâmica. Queremos salientar entretanto que, apesar das funções de configuração estarem embutidas na linguagem de programação, as ligações são feitas de forma totalmente explícita, e não de forma implícita como em várias outras linguagens. Isto, junto com outras características já mencionadas, faz com que num programa escrito em *MOD seja facilmente identificada a configuração do sistema, como acontece na nossa proposta.

A nossa proposta, além de não misturar as funções de programação, ligação e carregamento, se preocupa muito menos com a arquitetura física na qual o programa será executado, e muito mais com as facilidades de estruturação do programa. Por causa disto, fornecemos uma única forma de comunicação, ao invés das várias permitidas por *MOD, que satisfaz todos os casos e permitimos uma estruturação hierárquica, que facilita a construção de sistemas grandes. A flexibilidade de poder definir instâncias de módulos e sub-configurações simplifica muito a construção do sistema a partir de módulos já compilados, o que torna a nossa proposta bem mais poderosa. Mesmo não tendo definido ainda a forma de especificar a configuração dinâmica, achamos que a nossa proposta pode ser estendida sem muitas dificuldades.

VII.3 LINGUAGEM SR ("SYNCHRONIZED RESOURCES")

Analisaremos aqui as características da versão modificada da linguagem SR, descrita por ANDREWS e OLSSON em [6], que foi utilizada para implementar o sistema operacional distribuido SAGUARO [8]. Os objetivos principais que nortearam as modificações da linguagem foram os seguintes:

i) ser expressiva, no sentido de poder resolver problemas relevantes de maneira fácil;

ii) ter um entendimento e uso simples;

iii) ser eficiente tanto na compilação quanto na execução.

Os autores escolheram como componente fundamental o **recurso**, que é parecido ao módulo de Modula-2. O recurso tem uma parte de especificação, que define as operações providas pelo recurso, e a parte de implementação, que contém a declaração dos algoritmos correspondentes às operações. Um recurso é implementado por um ou mais processos que executam no mesmo processador. Os processos interagem através de operações que representam canais de comunicação parametrizados. Os processos pertencentes a um mesmo recurso podem compartilhar variáveis.

Pela sua experiência com o sistema SAGUARO, os autores concluiram que num sistema distribuido a maioria dos módulos são servidores, que gerenciam objetos tais como arquivos e atendem pedidos de clientes para ter acesso a esses objetos. Eles afirmam que os recursos de SR são mecanismos naturais para programar servidores. No caso de um servidor de arquivos, por exemplo, estes precisam ser abertos primeiro e isto é realizado por um processo separado. Dentro de cada módulo servidor, correspondente a um determinado arquivo, existe um processo para cada cliente que abriu o arquivo. Esta forma de tratamento de arquivos aumentam o paralelismo do sistema.

CRIAÇÃO E DESTRUIÇÃO DINAMICA DE RECURSOS

Na versão anterior de SR [4] os programas eram estáticos; o número de instâncias de recursos e de processos dentro de um recurso era fixo e especificado em tempo de compilação. Na nova versão [6] a linguagem é dinâmica e todos os recursos e

processos são criados explicitamente através da execução de comandos, para obter uma maior flexibilidade.

Os recursos são agora módulos parametrizados, a partir dos quais são criadas e destruidas instâncias durante a execução do programa. Esta modificação traz as seguintes vantagens em relação à versão anterior:

i) podem ser criadas múltiplas instâncias de um recurso a partir do mesmo componente de programa;

ii) é possível compor e testar diferentes configurações de um sistema, sem precisar recompilá-las;

iii) módulos de recursos já existentes podem ser reutilizados em programas novos, modificando somente os tipos dos parâmetros;

iv) os programas podem crescer ou diminuir durante a execução.

Para implementar este caráter dinâmico são acrescentados comandos de criação e destruição, e os recursos são agora referenciados indiretamente através de **direitos de acesso** ("capabilities"), conceito que já existia na versão anterior para permitir a definição de caminhos ("paths") de comunicação dinâmicos entre recursos estáticos. Com esta mudança os recursos agora não podem ser aninhados, mas também isto não se faz mais necessário, já que, se um recurso é implementado utilizando outro, ele pode, ou criar a instância que ele precisa, ou receber o direito de acesso passado para ele como parâmetro.

Cada instância de um recurso é alocada a um determinado nó físico ou máquina, através do seguinte comando:

cap_var:=**create** resource_identifier(actuals) **on** machine

Vemos que na criação já é definida a alocação física do módulo criado. Na execução deste comando, além de criar uma instância, é retornado um direito de acesso, que é uma variável do tipo **cap**. Ela será utilizada para invocar as operações exportadas pelo recurso, ou para destruir a instância através do seguinte comando:

destroy cap_var

Cada programa escrito em SR contém um recurso nomeado de principal ("main"). A execução do programa começa pela criação de uma instância deste recurso, que depois criará os outros que compõem o sistema. Naturalmente é necessário que já estejam carregados os módulos padrões dos tipos de recursos utilizados em cada máquina, para poder criar as instâncias correspondentes.

COMUNICAÇÃO E SINCRONIZAÇÃO

SR adaptou o **esquema híbrido de criação de processos** nos recursos: alguns processos são criados implicitamente e de forma fixa, em particular os que executam tarefas de apoio ("background"), e outros podem ser criados explicitamente.

Em relação às formas de comunicação e sincronização entre processos pertencentes a recursos diferentes, esta última versão optou por implementar **vários esquemas**. Por um lado existe a troca de mensagens síncrona e assíncrona, mas também é oferecida a chamada remota de procedimentos com duas implementações distintas. O programador pode escolher se um novo processo será criado especialmente para atender uma chamada de operação, ou se esta chamada será atendida através de "rendezvous" com um processo já existente.

A motivação que levou a estas escolhas é novamente o objetivo de fornecer o máximo de flexibilidade.

Foram acrescentadas também outras primitivas de comunicação para recepção assíncrona de mensagens, para terminação antecipada de chamadas, e para execução concorrente de comandos.

Para permitir que processos de um mesmo recurso compartilhem variáveis, é necessário ter mecanismos que garantam a exclusão mútua no seu acesso. Para isto foi escolhido o uso de semáforos.

TRATAMENTO DE FALHAS

Devido à preocupação com a confiabilidade dos sistemas distribuídos, e em particular do sistema SAGUARO, os autores definiram mecanismos para o tratamento de falhas. Depois de fazer uma discussão sobre os diferentes tipos de falhas e os diferentes

mecanismos para tratá-los, eles decidiram se preocupar somente com as falhas que podem ocorrer por colapso ou limitações de hardware, e não considerar as possíveis falhas de software, como já foi mencionado na seção II.3.4.

Eles consideram ter escolhido um enfoque intermediário, no qual o núcleo ("kernel") é encarregado de detectar as falhas, e o programador é responsável por tratá-los. Por exemplo, se um recurso detecta que outro falhou, pode ser apropriado criar uma nova instância deste último recurso. Para implementar este enfoque são oferecidos dois mecanismos básicos:

i) comandos de invocação e controle de recursos (**call**, **send**, **create** e **destroy**), que retornam o estado do recurso armazenando-o no direito de acesso utilizado no comando, tendo um conjunto de valores possíveis (**success**, **crash**, **nospace**, **terminated**, **undefined**);

ii) uma função booleana predefinida chamada **failed** que tem como argumento um direito de acesso ou o nome de uma máquina, e para a qual será associado um conjunto de comandos a serem executados, dependendo do seu resultado.

COMPARAÇÃO DE SR COM A NOSSA PROPOSTA

Nesta comparação a primeira diferença a ser colocada é a decisão dos autores de definirem uma nova linguagem, cuja definição e evolução foram dirigidas especificamente para o desenvolvimento do projeto SAGUARO. Portanto podemos notar que, apesar de serem levantados objetivos gerais, a linguagem ficou marcada pelas necessidades particulares do projeto mencionado.

A linguagem SR dá suporte à configuração dinâmica através de primitivas embutidas na própria linguagem. Não existe uma separação entre a programação em pequena escala e a programação em larga escala. A alocação física dos módulos do programa (recursos) também está embutida na única linguagem. Portanto não existe separação de funções, nem para a definição da configuração, nem para o mapeamento físico, em relação à programação em pequena escala.

Os autores defendem em [6] que todas as facilidades acrescentadas são simples, fáceis de implementar, e que não

acrescentam custo importante na execução. A ênfase maior na defesa da sua proposta é a flexibilidade da linguagem.

Apesar de um programa em SR ser composto a partir de um conjunto de módulos dos quais podem ser criadas várias instâncias, da mesma forma que faz a nossa proposta, não fica simples de identificar a configuração do sistema, em contraposição ao uso da nossa linguagem de configuração, que define claramente a composição da configuração que pode ser estruturada através de vários níveis hierárquicos. Não fica claro para nós como será definido o carregamento em cada nó físico dos tipos de módulos, a partir dos quais serão criadas as instâncias de forma dinâmica pelo módulo principal. É citado em [6] que estes módulos devem ser previamente carregados em cada nó físico.

Se, por um lado reconhecemos o grande potencial da configuração dinâmica, por outro lado, esta, por estar embutida dentro da programação em pequena escala, fica diluida, comparada com outras propostas, nas quais a definição das mudanças é feita de forma mais explícita, como na proposta de CONIC, por exemplo.

Utilizando o argumento deles, que nos sistemas distribuidos a maioria dos módulos são módulos do tipo servidor oferecendo serviços a clientes, não fica clara a importância de fornecer todas as combinações possíveis de mecanismos de comunicação e sincronização. Este mesmo argumento foi utilizado por nós, e por várias outras propostas apresentadas no Capítulo III, para a escolha de RPC para comunicação entre processos. A variedade de comunicação e sincronização e a variedade de implementações de RPC citadas, se por um lado aumentam a flexibilidade da linguagem, por outro devem provocar uma certa ineficiência.

A proposta deles, permitindo compartilhamento de variáveis com mecanismos próprios, dentro do recurso, e embutidndo a função de alocação na LPP, faz com que o sistema seja mais dependente da arquitetura física e menos transportável.

Temos que reconhecer que um ponto importante que não foi abordado ainda na nossa proposta é o tratamento de falhas. Neste sentido, mesmo oferecendo mecanismos simples e somente para falhas associadas ao hardware, SR tem se preocupado em dar algumas soluções, o que é necessário para garantir a confiabilidade de um sistema distribuido.

VII.4 LINGUAGEM ARGUS

A linguagem ARGUS, apresentada por LISKOV e SCHEIFLER em [97], foi projetada para implementar ambientes de programação distribuída, tendo em vista os seguintes requisitos a serem satisfeitos.

i) que o sistema possa continuar o seu funcionamento como um todo, mesmo na presença de falhas de nós ou da rede.

ii) poder fazer mudanças tanto físicas quanto lógicas de forma dinâmica sem parar o sistema.

iii) satisfazer a autonomia dos nós devido ao fato que a distribuição não só responde a considerações de eficiência, mas também, às vezes, a considerações políticas e sociológicas.

iv) garantir a distribuição de maneira modular para limitar os efeitos das mudanças.

v) tirar proveito da concorrência para aumentar a eficiência e diminuir o tempo de resposta.

vi) garantir a consistência do sistema.

Os autores apontam que o requisito mais difícil de ser satisfeito é o de consistência, que poucas linguagens atendem, e para o qual eles decidiram prover **atomicidade**. Este conceito vem sendo muito utilizado nas aplicações de bancos de dados, mas não foi integrado em linguagens de programação.

As propriedades, que caracterizam uma atividade atômica chamada de ação, são a sua indivisibilidade e a sua recuperação. A indivisibilidade é obtida exigindo que a execução da atividade não se sobreponha à execução de nenhuma outra atividade. A recuperação é implementada de forma que a atividade é executada totalmente, ou não é executada. Portanto seus objetos permanecem no estado inicial ou mudam para o estado final. Se acontecer uma falha durante a execução de uma atividade, é possível completar a atividade ou recuperar seus objetos no estado inicial.

As ações podem ser decompostas em entidades de forma estruturada hierarquicamente, chamadas de **ações aninhadas ou sub-ações**. Os autores apontam que a aplicação mais importante das

ações aninhadas é o mascaramento de falhas de comunicação. Portanto eles acreditam que a melhor forma de comunicação, num sistema distribuido, é através de chamada remota de procedimento, com a semântica denominada de no-máximo-uma-vez. Desta forma a mensagem é entregue e executada de uma vez, com exatamente uma única resposta recebida, ou a mensagem não é entregue e o cliente é informado.

No modelo por eles apresentado existem **dois tipos de ações: as sub-ações e as ações de alto nível**. Estas últimas correspondem às atividades que interagem com o ambiente externo, ou no caso de ações aninhadas de alto nível, às atividades que não devem ser invalidadas se os pais são abortados, As sub-ações, por outro lado, correspondem às atividades internas que realizam parte das interações externas.

CARACTERÍSTICAS DA LINGUAGEM

Nesta seção apresentaremos resumidamente as características da linguagem, algumas das quais já foram mencionadas no Capítulo V, na seção V.4.5.2.

ARGUS é baseada na linguagem sequencial CLU (LISKOV [95, 93]), que foi escolhida por dar suporte à construção de programas bem estruturados através de mecanismos de abstração, e por ser orientada a objetos. As características novas mais importantes que foram acrescentadas, são a implementação de guardiães, que representam os nós lógicos do sistema e a implementação de ações, que acabamos de apresentar.

Em ARGUS um programa distribuido é composto de um conjunto de **guardiães**. O guardião encapsula e controla o acesso a um ou mais recursos, por exemplo bases de dados ou periféricos. Ele torna os recursos disponíveis aos usuários provendo um conjunto de operações chamadas **manuseadores** ("handlers"), que podem ser chamados por outros guardiães para usar os recursos. O guardião executa os manuseadores, sincronizando-os e controlando seu acesso da forma necessária.

O guardião executa num único nó físico e pode sobreviver a colapsos do nó com uma probabilidade alta, e por isto os guardiães podem ser considerados flexíveis. Para satisfazer esta propriedade, o estado do guardião consiste de objetos estáveis e

objetos voláteis. Os objetos estáveis são armazenados em memória estável, e depois de acontecer um colapso de um nó físico, a linguagem dá suporte para recriar o guardião a partir dessa informação estável, com os últimos valores registrados. É dada partida a um processo do guardião para recriar os objetos voláteis, e, uma vez concluida esta restauração, o guardião pode retomar as suas tarefas principais e responder a novas chamadas de manuseadores. Estas chamadas são executadas utilizando o mecanismo de troca de mensagens, e, como os guardiães não compartilham variáveis, os argumentos dos manuseadores devem ser passados por valor. O atendimento das chamadas é feito criando um processo separado para cada chamada, o que permite aumentar o paralelismo na execução do guardião, que pode assim atender várias chamadas concorrentemente. O escalonamento da execução das chamadas dos manuseadores é feito pelo sistema e é transparente para o usuário.

Os guardiães são criados dinamicamente e o programador especifica o nó físico no qual o guardião é criado. O nome do guardião e os nomes de seus manuseadores podem ser comunicados nas chamadas de manuseadores. Uma vez recebidos, podem ser feitas chamadas para esse guardião, e estas são independentes da alocação física, o que facilita a reconfiguração do sistema.

Guardiães e manuseadores são abstrações da arquitetura física subjacente do sistema distribuido. **O guardião é o nó lógico do sistema**, e a comunicação entre guardiães através dos manuseadores é uma abstração da rede física. A maior diferença entre o sistema lógico e o físico é a confiabilidade

O guardião provê também **criadores** ("creators"), que são operações invocadas para criar novas instâncias de tipos de guardiães. Os nomes dos criadores são listados, junto com os dos manuseadores no cabeçalho do guardião. Os guardiães podem ser parametrizados, permitindo definir classes de abstrações relacionadas através de um simples módulo. Além de criar uma nova instância e de executar a inicialização das variáveis de estado do guardião, o criador tem essencialmente a mesma semântica que o manuseador. A chamada do criador é executada dentro de uma nova sub-ação do chamador.

O guardião contém um código de recuperação que será executado depois de um colapso, como uma ação de alto nível.

Depois do guardião ser criado ou recuperado com sucesso, ele executará seu código principal ("background") que contém tarefas periódicas ou contínuas. Esta seção não é executada como ação, embora geralmente ela cria ações de alto nível para executar as tarefas.

ARGUS, assim como CLU, provê **compilação separada de módulos** com verificação de tipo, feita no contexto de uma biblioteca de programa, que contém a informação das abstrações (tipos de guardiães).

Antes de criar uma instância de um guardião num nó, é necessário carregar seu código. Podem existir diferentes tipos de um mesmo guardião. Para montar o código imagem da definição de um guardião é necessário escolher a implementação dos dados, dos procedimentos e das abstrações de iteração que são utilizadas. Cada guardião é ligado e carregado separadamente. De fato, cada guardião é independente da implementação dos outros guardiães, porque o método de comunicação de dados entre guardiães é independente da implementação.

A verificação em tempo de compilação não afeta a **reconfiguração dinâmica**. Pelo fato de receber guardiães e manuseadores dinamicamente nas chamadas de manuseadores, um guardião pode comunicar-se com guardiães novos à medida que eles são criados e se tornam disponíveis. Isto permite mudar implementações de guardiães que estejam executando. O problema que os autores estão pesquisando é a estratégia de substituição a ser utilizada para não afetar os usuários desses guardiães.

A diferença fundamental desta linguagem em relação às outras é a preocupação com o **tratamento de falhas** e a consistência do sistema. Entretanto os autores destacam duas áreas que ainda não foram bem resolvidas. Uma é proteção, no sentido que a linguagem não permite expressar restrições em relação a onde e quando os guardiães podem ser criados. Este ponto foi em parte pesquisado por BLOOM em [23] e foi apresentado no Capítulo V. A outra área mencionada é o escalonamento do atendimento das chamadas dentro de um guardião que é transparente para o usuário. Os autores consideram que em algumas situações seria mais conveniente poder atribuir prioridades às chamadas.

COMPARAÇÃO DE ARGUS COM A NOSSA PROPOSTA

ARGUS tem algumas semelhanças com a nossa proposta, principalmente o fato de estar baseado numa linguagem já definida (CLU) que dá suporte à compilação separada.

A linguagem oferece uma estruturação modular e hierárquica através dos conceitos de guardiães, ações e sub-ações, diferente da nossa, já que os guardiães são criados e destruídos dinamicamente dentro de guardiães. Esta forma é parecida, em parte, com o modelo de SR em relação à criação e destruição dinâmica e a estruturação.

ARGUS não oferece uma linguagem separada de configuração, e assim como em SR as funções de criação e alocação física estão juntas e dentro da programação em pequena escala. Para isto devem estar carregados nos nós físicos os códigos dos guardiães a serem criados, e não fica claro como é definido que tipos de guardiães devem ser carregados em cada nó.

Assim, como na nossa proposta, em ARGUS é possível criar instâncias de tipos de guardiães, e, devido à configuração dinâmica de ARGUS, é possível substituir implementações de tipos de guardiães em tempo de execução.

O atendimento das chamadas dentro de um guardião é feito de forma concorrente como na nossa proposta, mas sem um número fixo de processos, ou seja, é criado um processo para cada chamada, sem limitações e com escalonamento de processos transparente para o usuário. Uma diferença importante reside no fato que em Modula-2 o usuário pode definir a política de escalonamento de processos mais apropriada a ser programada no nível do núcleo.

Em ARGUS a comunicação e sincronização entre processos é feita através do mecanismo de troca de mensagens, mas os autores sugerem como mecanismo mais adequado a RPC, por se assemelhar às chamadas de manuseadores.

A característica mais marcante de ARGUS é a implementação de ações atômicas para garantir a consistência do sistema programado nesta linguagem. Os autores enfatizam que com estas ferramentas uma RPC poderia ser executada como sub-ação para mascarar as possíveis falhas. Se por um lado é reconhecido o potencial do uso de ações atômicas, por outro a sua incorporação no nível da linguagem é questionada na literatura, e este tema será discutido

com mais detalhe posteriormente.

Insistimos em apontar que a falta de uma linguagem de configuração separada ocasiona para um programa uma dependência da arquitetura física. Mesmo sendo destacado que em ARGUS as chamadas não dependem da alocação física, e por isto a reconfiguração é facilitada, queremos enfatizar que pelo fato da criação de guardiães estar ligada à alocação física, existe uma dependência entre o programa e a arquitetura física.

Em ARGUS as conexões entre guardiães não são expressas de forma explícita. Elas são realizadas através da transmissão de interfaces de guardiães para outros guardiães. Desta maneira as conexões são implícitas e é dificultada a identificação da estrutura do sistema. Isto é consequência da falta de separação entre as funções de criação e ligação de guardiães, que correspondem à programação em pequena escala e à programação em larga escala, respectivamente. Em ARGUS, somente a lista dos nomes dos arquivos providos ao ligador representam algo parecido a uma especificação de configuração.

Como os guardiães são ligados separadamente, dois guardiães do mesmo tipo executando no mesmo sistema podem ter implementações diferentes, de forma análoga à que foi implementada na nossa proposta. O sistema ARGUS inclui um serviço de catálogo que pode ser utilizado pelos clientes para localizar outros serviços.

A linguagem ARGUS tem colocado como requisito prioritário garantir a consistência do sistema distribuido e neste sentido ela é uma linguagem pioneira. Continuam os estudos para incorporar formas seguras de implementar a reconfiguração dinâmica, como foi mostrado no Capítulo V, na seção V.5.2.

VII.5 PROPOSTA DE ADAPTAÇÃO DE MODULA-2 PARA SISTEMAS DISTRIBUIDOS

APRESENTAÇÃO DA PROPOSTA

Devido ao uso cada vez mais difundido de microcomputadores, e ao enorme interesse de construir computadores de processadores múltiplos para aplicações de tempo real, MELLOR, DUBERY e WHITEHEAD [111] decidiram estender a linguagem Modula-2 para ser

usada em arquiteturas distribuídas. A característica mais importante da arquitetura considerada é estar constituída por um conjunto homogêneo de processadores com comunicação rápida e sem memória compartilhada.

No sistema de programação definido por eles, a alocação das partes do programa na arquitetura física é realizada de forma iterativa (isto é, alocação, execução, realocação) e de forma independente do processo de codificação. Normalmente, portanto, o **sistema é compilado de forma separada**, testando de forma independente cada módulo que não possua referências à distribuição.

O processo de ligação foi modificado, comparado com o utilizado para um programa a ser executado por um único processador, já que neste modelo os módulos precisam estar alocados aos diferentes processadores. O ligador foi dividido em dois programas separados: o construtor e o ligador/alocador. O **construtor** localiza e processa todos os módulos necessários para a ligação do programa distribuído, realizando aquelas funções do ligador que são independentes da alocação, e portanto precisam ser feitas uma única vez. O **ligador/alocador** aloca e liga os componentes gerando um arquivo de imagem do programa distribuído, que será então carregado na arquitetura correspondente. Portanto o programa distribuído é processado de forma centralizada para ser depois carregado na configuração distribuída. No caso de se querer introduzir mudanças, o programa pode ser novamente alocado e ligado sem precisar ter que ser recompilado.

Vamos analisar a seguir as principais características desta proposta.

A linguagem Modula-2 estendida utiliza, como mecanismo de comunicação e sincronização entre processos em nós físicos distintos, a **chamada remota de procedimento de forma transparente**, e portanto não modifica a linguagem, que mantém a mesma sintaxe para chamadas locais ou remotas. Por outro lado, não é permitido referenciar variáveis que estejam alocadas em outro nó, e é exigido que os módulos que precisem compartilhar dados estejam alocados no mesmo nó. Portanto, se existem variáveis exportadas, seus importadores serão alocados ao mesmo nó físico.

Além do objetivo de estender Modula-2 para rodar em

arquiteturas de multiprocessadores, os autores destacam um segundo objetivo que é a **separação (iterativa) das fases de codificação e alocação.** Já foi mencionado que a informação sobre a alocação dos módulos não faz parte do código fonte.

A unidade de alocação escolhida foi o **módulo global.** As três razões principais são as seguintes:

i) os procedimentos de um módulo precisam ter acesso eficiente aos dados locais compartilhados do módulo;

ii) muitas vezes os módulos representam monitores nos quais é preciso garantir a exclusão mútua no acesso dos processos. O fato do módulo ser alocado num único nó físico garante e reforça os argumentos apresentados.

Não foi considerada a possibilidade de alocar módulos locais, independentemente dos módulos nos quais eles são declarado, porque isto acrescentaria uma complexidade grande, decorrente da forma em que os módulos são ligados, que analisaremos a seguir.

Apesar do código fonte não conter informação sobre a alocação dos módulos, foi acrescentada a Modula-2 informação sobre as ligações entre módulos. Esta informação coloca restrições, no momento da ligação, sobre as possibilidades de alocação que o alocador precisa detectar e reforçar.

Existem dois tipos de ligações definidas: as relativas e as absolutas.

A ligação relativa corresponde à ligação entre módulos que exportam e importam objetos comuns. Este tipo de ligação é detectada pelo compilador, que gera informação nos arquivos para ligação, registrando as restrições de alocação. Para isto foi modificada a sintaxe da declaração de importação da seguinte forma:

<importação>=[FROM ident] IMPORT [TIGHT] identList;

A palavra reservada **TIGHT** pode aparecer na lista de importações somente dos módulos globais.

A ligação absoluta é a ligação entre um módulo e um nó físico; ela foi definida para atender a casos, tais como por exemplo, ter acesso direto aos dispositivos de entrada/saída ou

esperar interrupções específicas desse nó.

Para refletir esta ligação, foi modificada a sintaxe do cabeçalho da declaração do módulo da seguinte forma:

<Unidade de compilação> = Definition Module: [FIXED] [IMPLEMENTATION] ProgramModule.

Geralmente este tipo de módulo referencia variáveis com endereços absolutos e/ou contém comandos do tipo **IOTRANSFER**. A especificação do nó físico no qual o módulo será alocado será feita durante a alocação. Este tipo de módulo não pode ser realocado automaticamente.

Devido à presença, tanto das ligações relativas quanto das absolutas, pode acontecer que um programa seja compilado corretamente mas que não possa ser alocado. Isto mostra uma determinada dependência entre o programa e a arquitetura física na qual será executado.

Os autores destacam a vantagem do uso de RPC em relação ao paralelismo dos processos, que podem chamar procedimentos em vários outros nós, e em relação a mudanças, já que acontecendo uma realocação, o sistema reconhece se uma chamada que era remota antes, pode ser local depois da realocação, aumentando a eficiência da chamada. É mostrado ainda o uso de monitores, junto com RPC, para comunicação entre nós diferentes, que é mais natural em Modula-2 do que outros mecanismos, tais como canais ou "rendezvous", utilizados em sistemas distribuídos.

Eles frisam a importância de dar prioridade, no escalonamento de processos, àqueles que atendem chamadas remotas em relação aos que processam localmente.

A implementação, baseada no compilador original de Modula-2 de WIRTH, foi desenvolvida utilizando uma arquitetura contendo um processador mestre e vários escravos, ligados através de um barramento paralelo. O processador mestre funciona principalmente como controlador de comunicação entre os escravos e como interface com o próximo nível (o sistema supervisor). Durante a fase de ligação, o ligador detecta as chamadas remotas e modifica o código redirecionando as chamadas para o suporte de RPC. O sistema é então constituído de três níveis: os programas de aplicação, o sistema de RPC e o sistema de comunicação de

mensagens. Cada nível utiliza o nível logo abaixo.

É mencionado em [111] a existência na implementação de um programa alocador, responsável pela alocação interativa e iterativa dos módulos nos nós físicos.

COMPARAÇÃO COM A NOSSA PROPOSTA

Nesta extensão da linguagem Modula-2 para programação de sistemas distribuídos, existem duas diferenças particularmente importantes em relação à nossa proposta que são as seguintes. A primeira é a modificação do compilador padrão de Modula-2, que precisou ser realizada por ter incluído na sintaxe da linguagem as especificações das ligações, relativas e absolutas. A segunda é o fato de não ter uma linguagem de configuração definida. Infelizmente a única fonte de bibliografia que conseguimos consultar foi o artigo já citado [111], no qual só se faz referência à separação da alocação do código de programação, mencionando a existência de um programa alocador. Por outro lado, queremos lembrar que as ligações aparecem, em parte, no código de programação e portanto não existe uma separação total de funções.

Estas duas diferenças nos levam a concluir que esta proposta é menos transportável que a nossa, e que é mais dependente da arquitetura física.

Outras diferenças importantes a ressaltar são que, por um lado, dada a falta de uma linguagem de configuração, parece difícil imaginar a possibilidade de uma estruturação hierárquica do sistema, que consideramos uma característica fundamental na estruturação de sistemas grandes. Dado que Modula-2 só permite estruturação hierárquica no nível dos módulos, e sendo o módulo global a unidade de configuração, a estruturação hierárquica no nível do sistema só pode ser obtida através de uma linguagem de configuração. Pelo outro, a falta de conceito de tipo de módulo não permite a criação de várias instâncias de um módulo e a possibilidade de associar diferentes módulos de implementação a uma mesma interface.

É descrito o processo de configuração estática, mas é só insinuado o processo de configuração dinâmica ou de reconfiguração.

Por outro lado, existem algumas semelhanças, sendo que a

mais importante é a decisão de estender Modula-2 para ser utilizada num ambiente de programação distribuida. Existem outras, como, por exemplo, em relação à unidade de configuração, que nos dois casos é o módulo global. Logicamente, no nosso caso também, se um módulo contivesse módulos locais, não seria permitido a sua remoção ou adição. Eles também adotaram como mecanismo de comunicação e sincronização entre nós distintos a chamada remota de procedimento do tipo transparente. Neste caso, infelizmente, não é mencionada a forma de implementação, para poder compará-los.

Na nossa proposta também, os módulos que compartilham variáveis, através de declarações de importação e exportação, devem ser carregados no mesmo nó. Isto é levado em conta, pelo gerador de "stubs", que primeiro testa, na análise dos módulos de definição, se existe alguma importação ou exportação de variáveis, e, neste caso, passa para outro módulo imediatamente. Em outras palavras, o gerador de "stubs" não vai gerar "stubs" para módulos que compartilham variáveis; desta forma não serão permitidas chamadas remotas para esses módulos.

O projeto deles também teve a preocupação de separar a codificação dos módulos da alocação do programa na arquitetura física. Entretanto isto não foi conseguido totalmente, já que foram incluidas as ligações na linguagem de programação.

Por falta de maiores informações sobre a proposta, fica difícil fazer uma comparação mais profunda, mas a impressão é que a nossa proposta é mais poderosa, flexível e transportável.

VII.6 LINGUAGEM MESA E C/MESA

A linguagem de programação MESA suporta programação modular de forma a permitir que subsistemas sejam projetados separadamente, e depois combinados para formar um sistema de maneira segura. A definição de interfaces separadas e explícitas oferece uma maneira efetiva de comunicação entre programas e entre programadores.

A linguagem de configuração C/MESA descreve a organização de um sistema de forma hierárquica e controla o escopo das interfaces. Estas facilidades tiveram um impacto profundo na forma na qual são projetados os sistemas e são organizados os

projetos de desenvolvimento.

Analisaremos algumas das características de MESA, que serão necessárias para estudar depois as propriedades da linguagem C/MESA, que possui várias das propriedades já mencionadas das linguagens de interconexão modular. Esta linguagem, definida em 1979 por MITCHELL et alii [113], já foi intensamente utilizada, em particular para implementar o sistema operacional Pilot [130], cuja experiência é relatada em [91].

LINGUAGEM MESA

Em MESA programas grandes são construidos ligando conjuntos de módulos individuais. **O módulo é a unidade básica de compilação e também é a unidade auto-contida executável.** Existem dois tipos de módulos. Os **módulos de definição** servem para especificar como as partes do sistema serão combinadas. Durante a compilação eles provêem o conjunto de definições comuns, que pode ser referenciado pelos outros módulos sendo compilados. **O módulo de programa** contém os dados e o código executável. Os módulos de programa se comunicam através de interfaces, e a existência desta separação provê a base para implementar a compilação separada dos componentes do sistema. Os módulos não podem ser aninhados, mas é possível, através da linguagem de configuração C/MESA, combinar conjuntos de módulos de forma hierárquica.

As interfaces geralmente contêm duas partes, a **parte estática**, que declara tipos e constantes, e a **parte de operações**, que define as operações (procedimentos e sinais) disponíveis aos clientes que importam estas interfaces. Somente os nomes e os tipos das operações são especificados na interface, e não as suas implementações. Esta parte da interface declara implicitamente um tipo de arquivo chamado de **arquivo de interface.**

O módulo que usa esta interface é chamado de **importador** de instância do arquivo de interface. Cada módulo precisa listar as interfaces que ele importa, e pode ser considerado como parametrizado por estas interfaces.

O módulo que implementa uma interface é chamado de **exportador.** Este módulo contém as declarações dos procedimentos e/ou sinais, cada um com o atributo PUBLIC, que correspondem aos procedimentos e/ou sinais definidos na interface. O compilador

garante que os tipos do exportador serão compatíveis com os tipos correspondentes do arquivo de interface, e com os tipos esperados pelos importadores da interface.

O código fonte de um dado módulo pode indicar ao compilador a possibilidade de incluir módulos compilados previamente. Isto é necessário nos seguintes casos:

i) quando o módulo precisar de símbolos definidos por outros módulos;

ii) para importar interfaces neles definidas;

iii) para referenciar instâncias deles, depois de serem carregadas, para serem inicializadas através do comando **START**;

iv) para criar novas instâncias; e

v) para ter acesso a seus dados.

Isto é feito em MESA através da declaração **DIRECTORY** colocada antes da declaração do módulo, por exemplo:

```
DIRECTORY
    SimpleDefs: FROM "simpledefs"
    StringDefs: FROM "stringdefs"
```

Analisemos agora, com um pouco mais de detalhe, como são declaradas as importações e as exportações nos módulos de tipo programa.

A lista de importações de um programa declara quais **arquivos de interfaces** o programa necessita e os associa com os módulos de definição, chamados **tipos de interfaces**. Arquivos de interfaces e tipos de interfaces são diferentes. Os nomes dos arquivos de interfaces são declarados na lista de importações do módulo do tipo programa. A declaração das importações tem a seguinte sintaxe:

```
IMPORTS    <nome de arquivo      :     <nome do tipo
            de interface>               de interface>
```

Por exemplo:

```
DIRECTORY Defs1:FROM "defs1"; Defs2:FROM "defs2";
Prog:PROGRAM IMPORTS IRec:Defs1,I2Rec:Defs2 =
   BEGIN ... END.
```

Quando é omitido o nome do arquivo de interface na lista da declaração **IMPORTS**, dando somente o nome do tipo da interface, é suposto que o nome do arquivo da interface é igual ao nome do tipo da interface. Por exemplo, escrever

```
Prog: PROGRAM IMPORTS IRec:Defs1,Defs2=....
```

é equivalente a escrever:

```
Prog: PROGRAM IMPORTS IRec:Defs1,Defs2:Defs2=...
```

Às vezes é necessário ter acesso a mais de uma instância de um arquivo de interface, no caso de importar dois módulos exportando o mesmo tipo de interface. Neste caso podem ser criadas várias instâncias da forma ilustrada pelo seguinte exemplo:

```
DIRECTORY SymDefs : FROM "SymDefs";
   PartOfCompiler : PROGRAM IMPORTS mainSym : SymDefs,
        auxSym : SymDefs=
   BEGIN ... END.
```

Qualquer módulo pode importar um módulo do tipo programa nomeando-o no seu diretório, por exemplo:

```
DIRECTORY XProg1:FROM "XProg1",XProg2:FROM "XProg2";
Prog : PROGRAM IMPORTS frame1:XProg1,XProg2=
   BEGIN ... END.
```

Podem ser usados XProg1 e XProg2 para declarar variáveis de programa do tipo **POINTER TO FRAME** [XProg1] e analogamente com XProg2. Portanto as declarações acima têm um efeito similar às seguintes declarações.

```
frame1 : POINTER TO FRAME [XProg1] = ....;
XProg2 : POINTER TO FRAME [XProg2] = ....;
```

Estas importações de programas são análogas às importações de arquivos de interface.

Um módulo pode exportar uma interface se ele provê procedimentos, sinais, etc., declarados com o atributo **PUBLIC**, cujos nomes e tipos casem com os elementos da interface do módulo de definição. O programa pode também exportar a si mesmo como parte da interface, se seu nome aparecer com o tipo de programa apropriado.

Uma característica particular da MESA, que em geral as outras linguagens de programação não possuem, é que a implementação de uma determinada interface pode ser realizada por

um conjunto de módulos de tipo programa, cada um implementando uma parte da interface, e não necessariamente por um único módulo. Por outro lado, um módulo de tipo programa pode também implementar mais de uma interface.

É importante destacar que a importação de uma interface, em tempo de compilação, não liga as referências aos procedimentos nos módulos importadores com os procedimentos implementados por um módulo exportador. Essa ligação é feita posteriormente, quando o módulo compilado for ligado com outros módulos compilados de tipo programa, para formar o sistema.

Depois da compilação, um programa contém um conjunto de arquivos de interfaces virtuais, um para cada interface importada, e um conjunto de arquivos exportadores, um para cada interface exportada. A ligação do conjunto total de módulos para formar o sistema envolve portanto a associação dos arquivos de interfaces virtuais com as interfaces exportadas, para todos os módulos do conjunto.

LINGUAGEM C/MESA

Uma descrição de configuração, que é um programa escrito em C/MESA, define de que maneira um conjunto de módulos escritos em MESA são ligados para formar uma configuração. A ligação é realizada compilando a descrição de configuração, e resulta numa descrição de configuração binária, chamada BCD. A BCD mais simples e atômica é o módulo de código objeto de um módulo escrito em Mesa. Portanto, o compilador, de Mesa produz a BCD simples, e o compilador de C/MESA produz BCDs mais complexas. Uma vez criada a BCD, ela pode ser carregada e executada. O caregamento é composto de duas fases. Na primeira é criada uma instância da configuração, alocando cada módulo atômico da BCD e criando espaço extra para o sistema MESA e para as ligações com outros módulos. A segunda fase do carregamento completa o processo de ligação dos componentes da configuração. Uma vez carregada a configuração, cada instância de módulo tem todas as suas interfaces ligadas. Agora é necessário inicializar cada instância executando o código de inicialização para as variáveis estáticas e seu código principal.

Para apresentar as características principais de C/MESA nos

basearemos no exemplo desenvolvido em [175] mostrando as partes essenciais, e, portanto, mais detalhes podem ser encontrados na referência citada.

O exemplo consiste de três módulos em MESA: uma interface (módulo **DEFINITIONS**), um implementador e um cliente dela (representados cada um por um módulo **PROGRAM**).

A interface chamada LexiconDefs, contendo três procedimentos, é definida pela Figura (VII.1):

```
LexiconDefs:DEFINITIONS=

BEGIN
   FindString:PROCEDURE [STRING] RETURNS [BOOLEAN];
   AddString:PROCEDURE [STRING];
   PrintLexicon:PROCEDURE;
END.
```

FIGURA VII.1 - DEFINIÇÃO DO MÓDULO DE INTERFACE LexiconDefs

O módulo de implementação para essa interface, chamado Lexicon, contém a declaração dos três procedimentos por ela definidos e pode ser resumido pela Figura (VII.2):

```
DIRECTORY
   IODefs:FROM "iodefs" USING [WriteLine],
   LexiconDefs:FROM "lexicondefs",
   StringDefs:FROM "stringdefs" USING [AppendString],
   SystemDefs:FROM "systemdefs" USING [AllocateHeapNode,
                                       AllocateHeapString];

Lexicon:PROGRAM
   IMPORTS SystemDefs, IODefs, StringDefs
   EXPORTS LexiconDefs =
   BEGIN
      .
      .
      .
   FindString:PUBLIC PROCEDURE [STRING] RETURNS [BOOLEAN]
      BEGIN

      END;
   AddString:PUBLIC PROCEDURE [STRING] =
      BEGIN

      END;
   PrintLexicon:PUBLIC PROCEDURE =
      BEGIN

      END;
END;
```

FIGURA VII.2 - DEFINIÇÃO DO MÓDULO DE IMPLEMENTAÇÃO Lexicon

Podemos notar que este módulo, além de exportar a interface LexiconDefs, importa três interfaces denominadas SystemDefs, IODefs e StringDefs. As cláusulas **USING** na declaração **DIRECTORY**

indicam a que procedimentos de cada interface se poderá ter acesso neste módulo.

O módulo cliente da interface, denominado LexiconClient é ilustrado resumidamente pela Figura (VII.3). Ele provê uma interface simples de terminal para o usuário poder testar Lexicon, chamando os procedimentos por ele implementados, e importa as duas interfaces IODefs e LexiconDefs.

```
DIRECTORY
   IODefs:FROM "iodefs" USING [CR,ReadChar,ReadLine,WriteChar,
                               WriteLine],
   LexiconDefs:FROM "lexicondefs" USING [AddString,FindString,
                               PrintLexicon];
   LexiconClient:PROGRAM IMPORTS IODefs,LexiconDefs =
      BEGIN
          .
          .
          .
      END.
```

FIGURA VII.3 - DEFINIÇÃO DO MÓDULO CLIENTE LexiconClient

Uma descrição de configuração possível, para executar o módulo LexiconClient, é a ilustrada pela Figura (VII.4). Esta configuração liga os três módulos apresentados, e outros necessários também, para formar uma configuração completamente auto-contida. São colocados à direita comentários em relação às importações e exportações que são casadas dentro desta configuração.

```
Config1:CONFIGURATION
   CONTROL LexiconClient=
BEGIN
 Fsp;           --                                  EXPORTS SystemDefs
 IOPkg;         --                                  EXPORTS IODefs
 Strings;       --                                  EXPORTS StringDefs
 Lexicon;       -- IMPORTS SystemDefs,              EXPORTS LexiconDefs
                   IODefs,StringDefs
 LexiconClient  -- IMPORTS IODefs,LexiconDefs
END.
```

FIGURA VII.4 - DEFINIÇÃO DA CONFIGURAÇÃO Config1

As configurações não precisam ser auto-contidas, e elas podem importar interfaces que serão importadas pelos seus componentes. Desta maneira, podem ser construídos subsistemas com algumas importações não ligadas dentro do subsistema.

Podemos então apresentar uma outra configuração para o caso anterior, na qual será necessário importar algumas das interfaces, a partir de outras configurações. Esta nova versão da configuração anterior é ilustrada pela Figura (VII.5):

```
Config2: CONFIGURATION
   IMPORTS SystemDefs, IODefs, StringDefs
   CONTROL LexiconClient=
BEGIN
   Lexicon;
   LexiconClient;
END.
```

FIGURA VII.5 - DEFINIÇÃO DA CONFIGURAÇÃO Config2

A declaração **IMPORTS** na configuração satisfaz os mesmos propósitos que no módulo programa, vistos anteriormente, e o mesmo se aplica à declaração **EXPORTS**.

Nestes exemplos não apareceu a declaração **DIRECTORY** porque supomos que os nomes dos componentes são iguais aos nomes dos arquivos, por omissão ("default").

Vejamos agora como C/MESA provê meios de definir instâncias de tipos de interfaces, e de fazer associações com as instâncias de interfaces importadas através de parâmetros.

A declaração **IMPORTS** da Figura (VII.5) poderia ser substituida por

```
IMPORTS alloc:SystemDefs, io:IODefs, str:StringDefs
```

criando três instâncias de interfaces dos tipos correspondentes.

Por outro lado, a associação entre os tipos de interfaces importadas por Lexicon e as instâncias específicas pode ser feita da seguinte forma:

```
Lexicon [alloc,io,str];
```

Deve ser respeitada a ordem e o tipo das declarações de importação de Lexicon.

C/MESA oferece a possibilidade de expressar a instância do tipo de interface correspondente a uma instância do módulo de sua implementação da seguinte forma:

```
lexRec:LexiconDefs <-- alex:Lexicon [alloc,io,str];
```

ou

```
lexRec <-- alex:Lexicon [alloc,io,str];
```

A primeira expressão está indicando que é definida uma instância chamada lexRec do tipo de interface LexiconDefs, exportada pela instância denominada alex do tipo Lexicon, com as importações definidas pelos parâmetros entre colchetes. Na segunda expressão foi retirado o tipo LexiconDefs por estar implícito na declaração de exportação de Lexicon.

Incorporando estes conceitos, a definição de configuração Config2 pode ser reescrita da maneira ilustrada na Figura (VII.6).

```
Config3:CONFIGURATION
    IMPORTS alloc:SystemDefs, io:IODefs,str:StringDefs
    CONTROL lexClient=
BEGIN
    lexRec:LexiconDefs <-- alex:Lexicon [alloc,io,str];
    lexClient:LexiconClient [io,lexRec];
END.
```

FIGURA VII.6 - DEFINIÇÃO DA CONFIGURAÇÃO Config3

Esta configuração define uma instância lexClient do tipo LexiconClient, associada a instâncias de interfaces io e lexRec correspondentes ao módulo de tipo IODefs e ao módulo de interface de tipo LexiconDefs, exportado pela instância alex do tipo de módulo Lexicon. Este último tipo é definido pelas importações de instâncias de módulos alloc, io e str, correspondentes aos módulos de tipo SystemDefs, IODefs e StringDefs, respectivamente. Podemos então notar que, nesta configuração, são criadas instâncias de módulos de programa e de módulos de interface, amarrando ao mesmo tempo as ligações entre as importações e exportações; desta forma **as funções de criação de instâncias e de ligação não são separadas.**

Queremos salientar que, sempre que forem omitidos nomes na criação de instâncias de módulos de programas ou módulos de interfaces em C/MESA, serão assumidos por omissão os nomes dos tipos correspondentes.

Existem outras duas características poderosas que oferecem MESA e C/MESA que vamos apresentar a seguir.

A primeira é a possibilidade de **um componente poder exportar mais de uma única interface.** Por exemplo, se existir um módulo de programa StringsAndIO que exporta as duas interfaces StringDefs e IODefs, podemos reescrever a configuração Config2 da forma ilustrada pela Figura (VII.7).

```
Config4:CONFIGURATION
   IMPORTS alloc:SystemDefs
   CONTROL LexiconClient=
BEGIN
   [str:StringDefs, io:IODefs] <-- StringsAndIo [ ];
   Lexicon [alloc,io,str],
   LexiconClient [io,LexiconDefs];
END.
```

FIGURA VII.7 - DEFINIÇÃO DA CONFIGURAÇÃO Config4

As interfaces exportadas são associadas através da instanciação do módulo StringsAndIO, declarando seus tipos também.

A segunda característica se refere ao fato de uma interface exportada ser o resultado da contribuição de um conjunto de componentes. Por exemplo, vamos supor que Lexicon é dividido em dois módulos LexiconFA e LexiconP, com LexiconFA provendo os procedimentos FindString e AddString, e LexiconP o procedimento PrintLexicon. Cada um exporta LexiconDefs, mas nenhum implementa completamente a interface. Entretanto, LexiconClient enxergará uma única interface na definição de configuração ilustrada pela Figura (VII.8).

```
Config5: CONFIGURATION
   IMPORTS SystemDefs, IODefs, StringDefs
   CONTROL LexiconClient=
BEGIN
   lexRec:LexiconDefs <-- LexiconFA [ ];
   lexRec <-- LexiconP [ ];
   LexiconClient [IODefs,lexRec];
END.
```

FIGURA VII.8 - DEFINIÇÃO DA CONFIGURAÇÃO Config5

Neste exemplo vemos que num primeiro passo é criada uma instância de módulo de interface denomidada lexRec, do tipo LexiconDefs, e que por omissão incorpora a interface de LexiconFA. No passo seguinte é feita a fusão com a interface exportada por LexiconP, de maneira que a instância lexRec criada esteja composta agora pela combinação das duas interfaces.

Outra propriedade, já mencionada, de C/MESA é a possibilidade de definir configurações aninhadas, da mesma forma que em Mesa são definidos procedimentos locais dentro de outros procedimentos. Essas configurações podem ser instanciadas e parametrizadas, e podem exportar interfaces, da mesma forma que qualquer configuração. O aninhamento de configurações, além de

permitir definir níveis hierárquicos de configurações, permite esconder algumas interfaces exportadas por componentes de uma configuração. As regras de escopo para as exportações e importações de interfaces definem a sua visibilidade.

COMPARAÇÃO ENTRE MESA E C/MESA E O NOSSO AMBIENTE DE PROGRAMAÇÃO DISTRIBUIDA

O sistema composto pelas linguagens MESA (LPP) e C/MESA (LPL) foi analisado com muito detalhe por ser um sistema completo e bem definido, e o seu estudo forneceu contribuições importantes para o nosso trabalho.

A primeira semelhança importante entre os dois trabalhos é o fato deles estarem baseados numa única linguagem de programação (LPP) com estrutura modular e permitindo compilação separada. Os dois trabalhos definem também uma linguagem de configuração separada (LPL), para combinar os módulos a fim de compor o sistema. Queremos entretanto ressaltar que MESA e C/MESA foram definidas para um único processador, isto é, não consideram a programação distribuida, e por causa disto não se preocupam com a alocação física dos módulos.

A linguagem MESA, comparada com Modula-2, apresenta uma série de características que a tornam bem mais poderosa e que contribuem também para a grande flexibilidade oferecida por C/MESA.

Tanto em MESA quanto em Modula-2, a comunicação entre módulos é do tipo procedimental. Mas no caso de MESA, as interfaces que são as responsáveis pela comunicação entre módulos, são módulos separados dos módulos de implementação, como foi apresentado anteriormente. Em Modula-2 a interface, mesmo sendo declarada separadamente do módulo de implementação, está diretamente ligada a ele, já que precisa ter o mesmo nome e só pode existir um único módulo de implementação para cada interface. Portanto este módulo implementa a interface inteira e de forma única, de maneira que existe um único tipo de interface e módulo de implementação associado. Em Mesa a interface tem um tipo definido. Mas o módulo de implementação, declarado separadamente e com nome diferente, pode implementar a interface toda, ou parte dela, ou várias interfaces ou partes de interfaces

diferentes. Por outro lado, podem existir vários módulos de implementação de tipos diferentes que implementem o mesmo tipo de interface. É na fase de configuração que serão feitas as correspondências entre as interfaces e os módulos de implementação. Estas características tornam a linguagem MESA bem mais poderosa e flexível. Para poder implementar algumas destas características foi acrescentado a Modula-2 a noção de tipo de módulo.

As duas linguagens utilizam também uma declaração parecida para declarar as importações de interfaces e as exportações dos módulos de implementação. No sistema MESA (composto pelas duas linguagens), a sintaxe e a semântica da declaração são idênticas tanto no nível de programação (MESA) quanto no nível de configuração (C/MESA). No nosso ambiente isto não ocorre e a sintaxe e semântica das delcarações do sistema MESA são mais parecidas com a forma definida no nível de configuração. Somente neste nível o nosso ambiente pode considerar várias instâncias de módulos e vários tipos de módulos de implementação para uma mesma interface.

Em MESA existe a declaração **DIRECTORY** que indica ao compilador a inclusão de módulos compilados previamente a serem utilizados na definição de um módulo, e explicita em parte o contexto no qual esse módulo está sendo definido. Esta declaração é utilizada em C/MESA também. No nosso caso foi definida uma declaração equivalente, denominada **USE** que define totalmente os tipos de módulos de implementação, que são usados para criar instâncias locais, algumas das quais podendo ser exportadas, e as interfaces que são importadas, na definição de uma sub-configuração. A declaração só é utilizada no nível de configuração para definir totalmente o contexto da sub-configuração.

Comparando as linguagens de configuração podemos destacar duas diferenças importantes.

Em C/MESA são criadas diferentes instâncias de tipos de interfaces de módulos e de módulos de implementação, da forma já mostrada na apresentação da linguagem e citada anteriormente. A flexibilidade para criação de instâncias de tipos é bem maior, além da separação total entre módulos de interfaces e módulos de implementação, que permite todas as formas de combinação já

citadas. Na nossa linguagem de configuração, como já foi mencionado, só podem existir diferentes tipos de módulos de implementação para uma determinada interface.

A segunda diferença consiste na forma de fazer as associações entre as interfaces, os módulos de implementação, e os módulos que importam essas interfaces. Em C/MESA essa associação pode ser feita de forma parametrizada, incluindo regras no caso de omissão de nomes de instâncias, e isto causa a não separação das funções de criação e de ligação de instâncias. No momento da criação de uma instância são definidas as suas ligações com os outros módulos de forma implícita, através da definição dos valores reais dos parâmetros. Na nossa linguagem de configuração estas funções estão totalmente separadas. Por um lado, através da declaração **CREATE** são criadas as instâncias de módulos ou sub-configurações, explicitando a sua alocação lógica, e por outro são definidas as suas ligações através da declaração **LINK** de forma explícita, como foi apresentado no Capítulo anterior. Isto permite modificar as ligações mais facilmente, e no caso de estender o modelo de configuração estática para suportar a forma dinâmica também, não será mais necessário destruir as instâncias criadas.

As duas linguagens de configuração permitem a definição de diferentes níveis hierárquicos, o que facilita a construção de sistemas grandes além de determinar a visibilidade das interfaces pelas regras de escopo.

Os dois sistemas de programação permitem um controle rígido de tipos tanto na fase de compilação quanto na fase de definição de configuração, diminuindo expressivamente os controles a serem realizados em tempo de execução.

VII.7 MÉTODO MASCOT3

Mascot é um método para projetar e implementar sistemas concorrentes grandes especialmente voltados para a área de tempo real. A partir da primeira versão que foi definida durante os anos 70 por SIMPSON e JACKSON [150] houve uma série de modificações e evoluções que produziram as duas versões seguintes Mascot2 (SIMPSON [151]) e Mascot3 (BATE [16]). Esta última será analisada e comparada com a nossa proposta.

O método provê uma linguagem de desenvolvimento e uma notação gráfica que podem ser utilizadas, tanto para a fase de projeto do sistema de forma estruturada e hierárquica, quanto para a sua manutenção. Cada uma, a linguagem e a notação gráfica, podem ser derivadas uma a partir da outra. Mascot3 foi desenvolvido para ser independente da linguagem de programação utilizada na implementação do sistema, e da configuração física na qual o sistema será executado. A modularidade oferecida por Mascot facilita enormemente a fase de verificação do sistema.

Mascot é caracterizado pelo uso de uma **rede gráfica de fluxo de dados** como meio de expressar a estrutura do software, combinado com um método de desenvolvimento sistemático que garante que esta estrutura seja refletida com exatidão no software operacional resultante. Esse diagrama de fluxo de dados provê visibilidade do projeto e oferece uma visão estática do software, cuja validade é protegida por um esquema rígido de criação e destruição de processos em tempo de execução.

Nesta seção, nos referimos a esta última versão utilizando simplesmente o nome de Mascot. Uma das principais características de Mascot é a **concorrência**. Nele **um sistema é definido de forma hierárquica**, como um conjunto de processos paralelos que cooperam entre si. No nível mais alto da hierarquia essas linhas paralelas de execução são agrupadas em unidades chamadas **subsistemas**. O método Mascot permite fazer uma análise descendente ("top-down"), especificando progressivamente os subsistemas, separando-os em unidades cada vez menores até os níveis mais baixos, onde são identificadas as linhas de execução de forma individual. Estas unidades de concorrência indivisíveis são chamadas de **atividades** e elas são executadas num ambiente padrão provido por um **software de contexto**, cujas funções são implicitamente disponíveis ao software de aplicação. A interface entre o contexto e o software de aplicação é chamado de **interface de contexto**.

A outra característica importante de Mascot é a versão do fluxo de dados que o método fornece. Em cada nível da hierarquia, o subsistema é representado por uma rede através da qual os dados são transmitidos entre uma entidade ativa (subsistema ou atividade) e outra. No nível mais baixo, as fontes e reservatórios de informação correspondem à dispositivos de hardware, com os quais é preciso se comunicar, e Mascot define

elementos de software específicos chamados **servidores**.

REPRESENTAÇÃO GRÁFICA

Vamos apresentar primeiro, resumidamente, a notação gráfica oferecida por Mascot através do exemplo mostrado em [16]. Este exemplo é apresentado na forma de uma hierarquia de níveis descendentes, e o nível mais alto é representado pela Figura (VII.9).

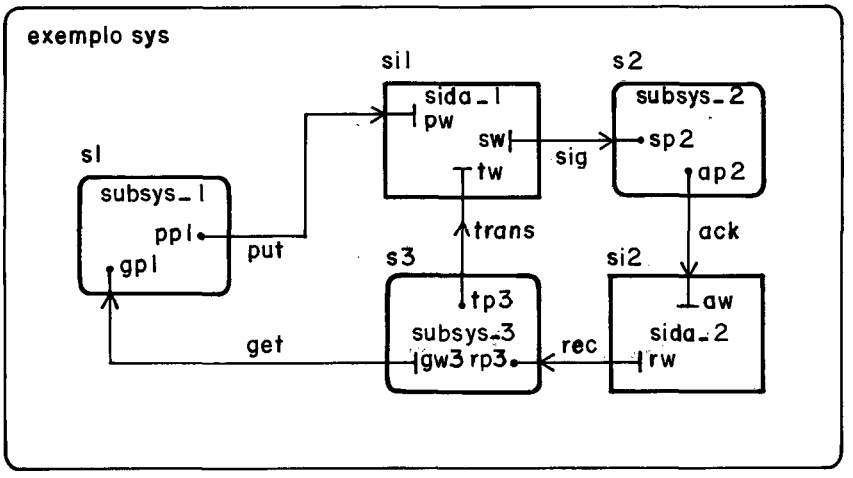

FIGURA VII.9 - EXEMPLO DO SISTEMA sys

O nível mais alto é representado por um retângulo de bordas arredondadas e está composto de três **subsistemas**, simbolizados da mesma maneira que o sistema, representando suas partes ativas que executam em paralelo. Os outros dois elementos representados por retângulos representam as **áreas de inter-comunicação de dados (IDA)**, através das quais os subsistemas se comunicam e sincronizam. Existe em Mascot então uma **separação nítida entre unidades ativas e passivas**. As linhas com setas indicam as ligações dos subsistemas com as IDA na rede e representam o fluxo de dados, chamadas também de **caminhos** ("paths"). Estas linhas ligam **portas** de entidades ativas, representadas por pontos, com **janelas** das entidades passivas representadas por tracinhos, e a direção do fluxo de dados é dada pela seta da ligação. Existem casos, como o do subsistema subsys_3, que contém portas também

para ligar o subsys_3 com o subsys_1 diretamente, sem passar por uma IDA.

Estes conceitos de portas e janelas serão tratados com mais detalhe posteriormente.

Vejamos agora com mais pormenores os componentes do subsistema subsys_3 representados pela Figura (VII.10).

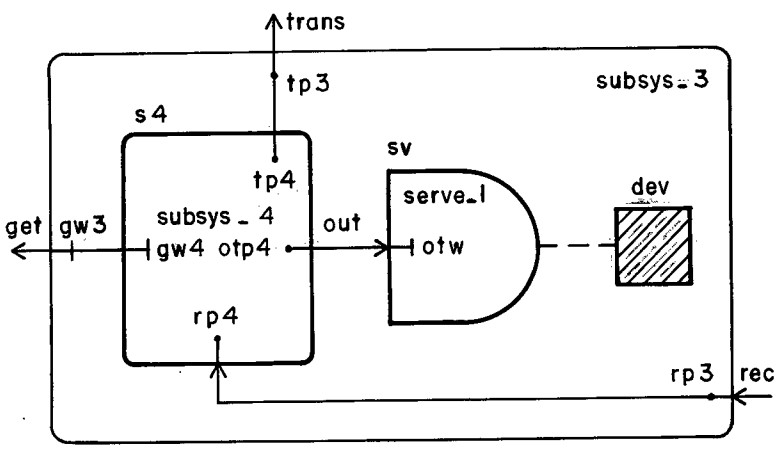

FIGURA VII.10 - REPRESENTAÇÃO DO SUBSISTEMA subsys_3

Podemos constatar que o subsistema subsys_3 está composto por um subsistema subsys_4, que se comunica com um servidor representado pelo símbolo D, ligado a um dispositivo, representado por um pequeno retângulo achureado, através de uma linha de tracinhos. Vemos que subsys_3 contém duas portas e uma janela para se comunicar com os outros componentes do sistema.

Continuando com o detalhamento descendente, podemos ilustrar a composição de subsys_4 pela Figura (VII.11).

Este subsistema está composto pelos elementos mais básicos, que são as **atividades** representadas por círculos, e os **canais** ("channels") e **reservatórios** ("pools"), que são os elementos básicos das IDA. Estes dois tipos de IDA possuem representações diferentes, que são variações dos retângulos. O canal, representado por um retângulo com dois lados opostos sobressalentes, se caracteriza pela operação de leitura

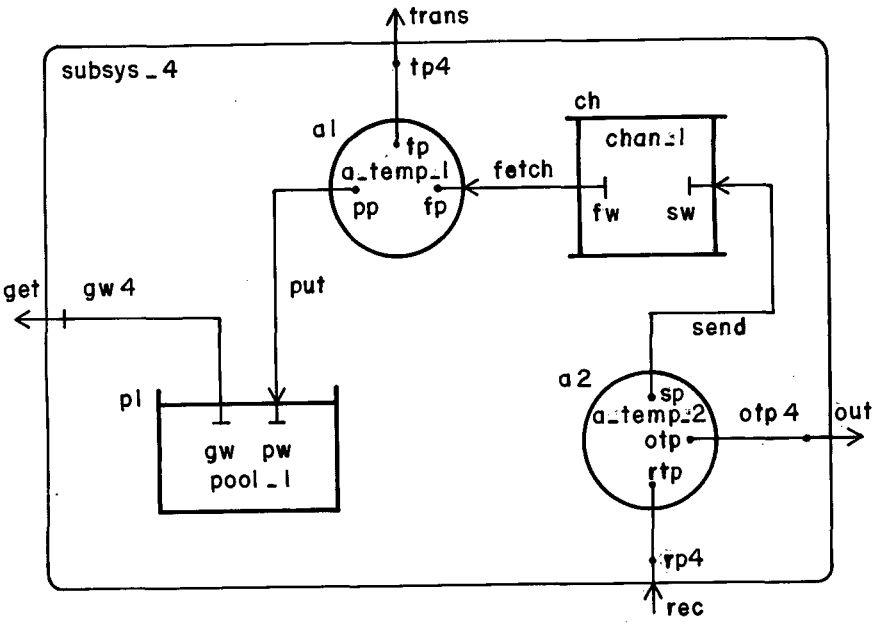

FIGURA VI.11 - REPRESENTAÇÃO DO SUBSISTEMA subsystem_4

destrutiva; os dados que fluem através dele são temporariamente acomodados na memória interna, que pode ficar cheia como resultado de operações repetidas de escrita, ou vazia como resultado de operações repetidas de leitura. No caso do reservatório, a operação que o caracteriza é a escrita destrutiva; ele é representado pela estrutura ilustrada na Figura (VII.11). Seu conteúdo consiste numa coleção de variáveis, para as quais são atribuídos valores iniciais quando o sistema inicia a execução, e que são depois consultadas e atualizadas.

Analisemos agora com mais detalhe o modelo de comunicação definido por Mascot. Para isto notemos que em subsys_4 a atividade a1 se comunica com a atividade a2 através do canal conectado entre as duas. Este caso ilustra um produtor (a2) fornecendo informação a um consumidor (a1). O canal funciona como depósito temporário ("buffer") e ele contém dois procedimentos, um para adicionar um ítem ao depósito, e outro para retirar um ítem de forma análoga. Estes procedimentos, chamados de acesso, são encapsulados pela IDA e estão disponíveis seletivamente para as atividades, através do conceito de janelas (as duas fw e sw).

Podemos notar também que estas portas do canal estão ligadas às atividades através de portas, fp e sp, pertencentes a a1 e a2 respectivamente, e as setas das linhas de ligação indicam a direção do fluxo de dados.

Podemos notar que as conexões, que não são casadas dentro do subsistema, são utilizadas para serem ligadas com portas e janelas (para o exterior) que pertencem a ele, para permitir sua ligação posterior a outros subsistemas ou IDAs. No caso de subsys_4, ele contém uma janela (gw4) e três portas (tp4, otp4 e rp4) para serem ligadas externamente.

Para completar a descrição da representação gráfica de Mascot, mostraremos como uma atividade, mesmo sendo um elemento básico, pode ser ainda decomposta num conjunto de componentes sequenciais. Para ilustrar estes conceitos, mostramos na Figura (VII.12) a decomposição da atividade a1 num elemento principal chamado de **raiz** e outros três elementos sequenciais com as suas relações de interdependência.

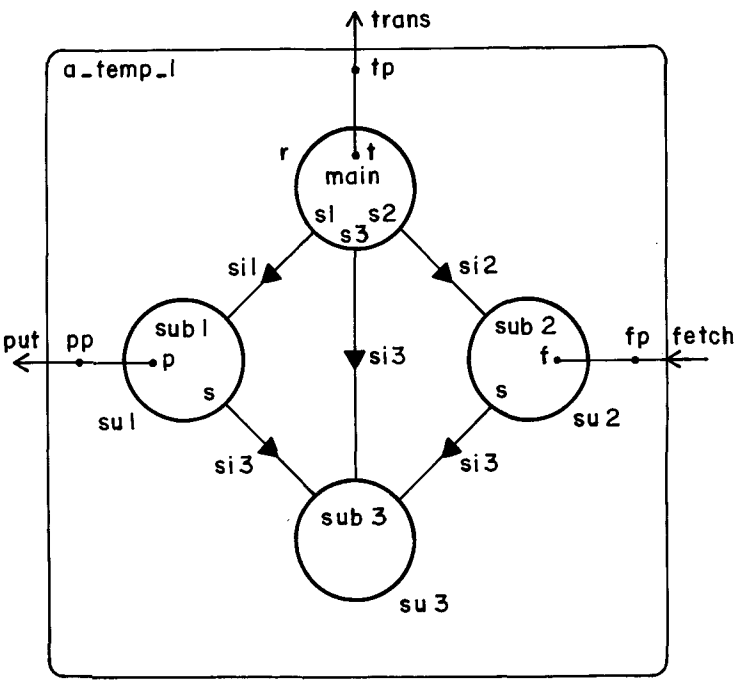

FIGURA VII.12 - DECOMPOSIÇÃO DA ATIVIDADE a1

LINGUAGEM DE ESPECIFICAÇÃO DA CONFIGURAÇÃO

Depois de ter analisado a representação gráfica do método Mascot vejamos qual é a representação textual correspondente, na linguagem de especificação da configuração.

Este estudo será feito de maneira oposta ao anterior, isto é, de forma ascendente ("bottom-up"), analisando primeiro a definição dos elementos básicos.

Existe aqui, de forma análoga à de Mesa, uma separação nítida entre as definições das interfaces das IDA e as suas definições.

Por exemplo, em relação a chan_1 ilustrado na Figura (VII.11), são declaradas duas interfaces de acesso, cada uma correspondente a uma janela da forma ilustrada pela Figura (VII.13).

```
ACCESS INTERFACE send;
   WITH flow_data;
   PROCEDURE insert (item:flow_data);
END.

ACCESS INTERFACE fetch;
   WITH flow_data;
   PROCEDURE extract (VAR item : flow_data);
END.
```

FIGURA VII.13 - DEFINIÇÃO DA INTERFACE DE chan_1

A declaração WITH é parecida com o USE da nossa proposta, e é utilizada para indicar que tipos de dados estão sendo utilizados. Estes tipos de dados globais são declarados através de DEFINITION. Podemos notar que as interfaces foram declaradas por separado, cada uma contendo somente um procedimento (por acaso, neste exemplo específico), e vemos que os nomes das interfaces não coincidem com os nomes dos procedimentos, apesar de serem os únicos elementos das interfaces.

A definição do chan_1 é ilustrada pela Figura (VII.14).

```
CHANNEL chan_1;
   PROVIDES sw:send;
            fw:fetch;

ACCESS PROCEDURE insert (item:flow_data);
      .
      .
END.

ACCESS PROCEDURE remove (VAR item:flow_data);
      .
      .
END.

fw.extract = remove
END
```

FIGURA VII.14 - DEFINIÇÃO DE chan_1

Podemos notar que esta definição exporta os dois procedimentos declarados nas interfaces através da declaração PROVIDES, que define nomes internos do tipo das interfaces. A declaração PROVIDES lista as janelas da IDA. Nesta definição estão declarados os corpos dos procedimentos das interfaces, um com o mesmo nome utilizado dentro da sua (insert), mas o outro com nome diferente. A associação entre os nomes é feita através da declaração de equivalência no final da definição de chan_1. Podemos concluir que esta definição de canal contém a definição da implementação de duas interfaces, e que a correspondência de nomes é feita de duas maneiras diferentes, determinando a associação entre a interface e a sua implementação.

```
ACTIVITY a_temp_2;
   REQUIRES sp:send; otp:out; rp:rec;
   VAR
      val:flow_data;
   BEGIN
      .
      .
   sp.insert(val);
      .
   END.
END.

ACTIVITY a_temp_1;
   REQUIRES fp:fetch; tp:trans; pp:put;
   VAR
      next:flow_data;
   BEGIN
      .
      .
   fp.extract(next);
      .
   END.
END.
```

FIGURA VII.15 - DEFINIÇÃO DAS ATIVIDADES a1 e a2

Vejamos agora a definição das atividades que chamam estas interfaces. Elas precisam importar os procedimentos a serem chamados e fica clara a **separação entre os componentes passivos e os ativos**. Resumidamente e de forma parcial, a Figura (VII.15) acima, ilustra as declarações das atividades.

Cada atividade, através da declaração **REQUIRES**, importa as interfaces necessárias, cujos procedimentos associados serão chamados. Vemos portanto que nesta declaração, que corresponde à lista das portas de cada atividade, aparecem as portas que serão posteriormente ligadas a chan_1, a pool_1 e às portas do subsistema subsys_4.

Dentro de cada atividade só aparecem as chamadas às interfaces send e fetch. As associações entre as interfaces importadas e as suas implementações são feitas no nível seguinte de configuração.

Passemos agora para o nível seguinte da hierarquia para analisar a definição de subsys_4, composto pelas duas atividades, o canal e o reservatório, ilustrado pela Figura (VII.16).

```
SUBSYSTEM subsys_4;
   PROVIDES gw4:get;
   REQUIRES rp4:rec;
            otp4:out;
            tp4:trans;
   USES pool_1,chan_1,a_temp_1,a_temp_2;
      POLL p1:pool_1;
      CHANNEL ch:chan_1;
      ACTIVITY a1:a_temp_1 (fp=ch.fw,
                            tp=tp4,
                            pp=p1.pw);
      ACTIVITY a2:a_temp_2 (sp=ch.sw,
                            otp=otp4,
                            rp=rp4);
   gw4=p1.gw
END.
```

FIGURA VII.16 - DEFINIÇÃO DO SUBSISTEMA subsys_4

Nesta definição o conjunto de declarações de exportação e importação representam a parte de especificação do subsistema. O resto corresponde à parte de implementação do subsistema, que se inicia com a declaração dos tipos de componentes, utilizados para criar instâncias, através da declaração **USES** dentro deste

subsistema. A seguir são criadas as instâncias, uma para o canal, uma para o reservatório e uma para cada tipo de atividade. As conexões são estabelecidas através da associação dos parâmetros das atividades. **Não há então separação entre as funções de criação e ligação de instâncias** e de forma análoga a C/MESA. Os nomes das portas à esquerda da equivalência são análogos, neste contexto, aos parâmetros formais de um procedimento. Os parâmetros reais correspondentes especificam os pontos da rede para os quais cada porta é ligada.

Detalhando o exemplo da atividade a1 vemos as seguintes ligações:

porta fp <---> janela fw de ch
porta tp <---> porta tp4 de subsys_4
porta pp <---> janela pw de p1

Através destas associações são definidas as ligações do tipo porta-janela e porta-porta, mas a ligação do tipo janela-janela é realizada através da declaração de equivalência colocado na última linha da definição do subsys_4.

Para resumir este processo ascendente de especificação mostraremos a definição do último nível da hierarquia correspondente ao sistema, ilustrado pela Figura (VII.17).

```
SYSTEM sys;
   USES subsys_1,subsys_2,subsys_3,sida_1,sida_2;
   IDA si1:sida_1;
   IDA si2:sida_2;

   SUBSYSTEM s1:subsys_1 (pp1=si1.pw,
                          gp1=si3.gw3);

   SUBSYSTEM s2:subsys_2 (sp2=si1.sw,
                          ap2=si2.aw);

   SUBSYSTEM s3:subsys_3 (tp3=si1.tw,
                          rp3=si2.rw);

END.
```

FIGURA VII.17 - DEFINIÇÃO DO SISTEMA sys

Vemos que esta definição é feita de forma análoga às de nível mais baixo, salvo algumas pequenas diferenças. Aqui é definido o sistema com a palavra **SYSTEM** e os **IDA** não são mais diferenciados como antes entre **CHANNEL** e **POOL**.

Podemos destacar então alguns níveis de hierarquia específicos. No nível mais baixo um subsistema é formado por elementos do tipo **ACTIVITY**, **CHANNEL** e **POOL**. A partir deste nível, os outros níveis hierárquicos de subsistemas estão compostos por elementos do tipo **SUBSYSTEM**, **IDA** e opcionalmente **SERVER**. No último nível a hierarquia é definida pela palavra reservada **SYSTEM** e não contém declarações de importação e exportação. Nos outros níveis, tanto estas últimas declarações, quanto as de **USES** e a forma de ligações de portas e janelas, são feitas da mesma maneira.

Existe ainda a **possibilidade de decompor uma atividade em sub-componentes sequenciais** como foi ilustrado na Figura (VII.12) cuja definição é representada pela Figura (VII.18).

```
ACTIVITY a_temp_1:
   REQUIRES fp:fetch;
            tp:trans;
            pp:put;

   USES main,sub1,sub2,sub3;
      SUBROOT su3:sub3;
      SUBROOT su1:sub1 (p=pp,
                        s=su3);
      SUBROOT su2:sub2 (f=fp,
                        s=su3);
      ROOT r:main (t=tp,
                   s1=su1,
                   s2=su2,
                   s3=su3);
END.
```

FIGURA VII.18 - DEFINIÇÃO DA ATIVIDADE a1

Aqui também existe a parte de especificação, composta pelas declarações **REQUIRES**, e a parte de implementação, formada pelo resto das declarações.

Na definição de cada componente **SUBROOT** e **ROOT**, aparecem novas declarações tais como **GIVES** e **NEEDS** que têm funções parecidas com **PROVIDES** e **REQUIRES**, só restritas ao âmbito da atividade.

Pretendemos ter mostrado nestas duas últimas seções as características principais, tanto em relação à representação

gráfica quanto em relação à linguagem de especificação do método Mascot.

Existem outros conceitos que gostaríamos de apresentar antes de iniciar a comparação entre este método e o nosso ambiente de programação distribuida. Queremos destacar que tanto subsistemas quanto IDAs podem ser ligadas entre si diretamente. Para isto é necessário que os subsistemas possuam janelas para serem ligadas com portas de subsistemas, e que as IDA possuam portas para serem ligadas com as janelas de outra IDA. As ligações entre componentes distintos sempre são feitas entre uma porta e uma janela, e podemos também ligar várias portas a uma mesma janela. Isto equivale a dizer que várias entidades ativas podem usar a mesma interface de uma IDA. O contrário não é válido. Outro ponto importante a ser destacado é que uma janela pode corresponder a vários procedimentos e não necessariamente a um. Isto corresponde a uma declaração de **ACCESS INTERFACE** contendo várias declarações de procedimentos.

As ligações entre portas e janelas, chamadas também de caminhos, têm uma direção definida pelo fluxo de dados. Mas existem não só caminhos unidirecionais, como foram vistos nos exemplos apresentados; podem existir caminhos bidirecionais também.

COMPARAÇÃO ENTRE O MÉTODO MASCOT E A NOSSA PROPOSTA DE AMBIENTE DE PROGRAMAÇÃO DISTRIBUIDA

A primeira diferença fundamental é que Mascot é um método que possui uma representação gráfica, além de uma linguagem de especificação de configuração. Achamos que não seria muito complexo definir uma representação gráfica para a nossa linguagem, se isso for considerado um requisito fundamental.

A outra diferença importante é que Mascot não depende da linguagem de programação utilizada para programar os componentes. O nosso projeto é totalmente dependente da linguagem Modula-2. Analogamente à MESA, Mascot permite definir tipos de atividades, IDAs e subsistemas para poder criar várias instâncias de cada.

A linguagem Mascot, apesar de permitir a construção de uma hierarquia de vários níveis, contém três níveis bem delineados nos quais são definidos conceitos, e particularmente, palavras

chaves diferentes, o que faz com que a linguagem possua um vocabulário extenso. Dos três níveis, o mais baixo é o nível de **ACTIVITY**, no qual uma atividade pode ser decomposta nas suas partes sequenciais, denominadas de **SUBROOT** e **ROOT**, com palavras chaves específicas para as suas interações tais como **GIVES** e **NEEDS**. O nível seguinte é o **SUBSYSTEM** composto de elementos básicos tais como **ACTIVITY**, **CHANNEL** e **POOL**. A partir deste nível podem ser construidos vários níveis compostos unicamente de elementos do tipo **SUBSYSTEM**, **IDA** e **SERVER** servidores. Somente o último nível da hierarquia é chamado de **SYSTEM**.

Esta estruturação parece em parte com a nossa, mas a consideramos bem mais complexa pelas diferenças introduzidas em cada nível.

Existe em Mascot uma separação muito nítida entre os elementos ativos e os passivos. Se por um lado a separação nítida entre a declaração de interface e a sua implementação, mais parecida com MESA e C/MESA, oferece uma flexibilidade maior na separação das interfaces em várias partes e na facilidade de definir várias implementações para uma mesma interface, por outro notamos certo grau de complexidade.

Tanto em MESA e C/MESA quanto no nosso modelo, não existe essa separação de entidades ativas e passivas. Os módulos que implementam as interfaces de comunicação entre módulos não são necessariamente passivos, e o fato de juntar as estruturas de dados com as operações a serem executadas sobre elas facilita a definição de tipos abstratos de dados. No caso de Mascot duas atividades nunca podem se comunicar diretamente, e é sempre necessário ter um tipo de IDA no meio. A diferenciação de tipos de IDA complica ainda mais esta estruturação.

Tanto em MESA e C/MESA quanto na nossa proposta a intercomunicação entre módulos é procedimental. Em Mascot, mesmo sendo a intercomunicação também procedimental, foram acrescentados os conceitos de portas e janelas, que ao meu ver complicam a compreensão e construção do sistema, em particular porque as ligações entre eles podem ter diferentes direções e não representam a direção na qual a chamada de procedimento é feita, tanto que é permitido definir caminhos até bidirecionais. As direções das linhas de ligação representam o fluxo de dados.

Outro ponto importante a ser colocado é a falta de separação

entre as funções de instanciação de atividades, IDAs e subsistema, e de ligação. Já mostramos que estas funções estão juntas e que as ligações são feitas de forma parametrizada ou através de declarações explícitas de equivalência. Fica colocada aqui a preocupação em relação a estes dois tipos de ligações, que facilitam a ocorrência de erros, além da falta de separação das funções mencionadas.

Nos artigos estudados é mencionada a possibilidade de configuração dinâmica mas não tem nenhuma descrição sobre a forma de fazê-la. Apesar da linguagem ter sido definida particularmente para sistemas grandes distribuídos de tempo real, é mencionado que não foi incluído nenhum esquema de mapeamento padrão no método.

Apesar de algumas críticas colocadas, reconhecemos que Mascot é um método muito poderoso e flexível, e que oferece maiores facilidades que a nossa proposta mas às custas de certa complexidade. Uma das ênfases colocada nos artigos sobre Mascot é a facilidade na verificação dos sistemas que ela oferece.

VII.8 PROJETO CONIC

Este projeto já foi mencionado em vários capítulos anteriores, por ter sido um dos trabalhos que norteou em parte a nossa proposta. Foi mostrada também a sua evolução e portanto nos concentraremos aqui principalmente nos aspectos que não foram abordados ainda.

Foram destacadas as características principais que listaremos resumidamente:

i) a linguagem de programação em pequena escala, chamada de CONIC, é uma extensão da linguagem Pascal, à qual foram acrescentados principalmente: o conceito de tarefa; primitivas de comunicação e sincronização entre tarefas; através de troca de mensagens síncronas e assíncronas; o conceito de portas locais de saída e entrada, para enviar ou retirar mensagens respectivamente.

ii) foi acrescentado o conceito de módulo, que contém a definição do contexto, a partir da declaração de uso de objetos comuns a vários módulos, e declarados em unidades de

definição distintas para permitir a compilação separada dos módulos.

iii) foi definida uma linguagem de configuração declarativa CONIC/C, que permite criar instâncias de módulos parametrizados de diferentes tipos, a partir dos quais será definida a configuração do sistema, ligando as portas de entrada e saída dos diferentes componentes, para definir o fluxo de informação entre eles.

iv) com a evolução do projeto CONIC foram introduzidas formas diferentes de estruturar a configuração hierarquicamente.

v) CONIC, desde o início, teve a preocupação de poder realizar configuração dinâmica usando a linguagem de configuração. Aqui também as idéias foram evoluindo ao longo do desenvolvimento do projeto e analisaremos com mais detalhe as últimas versões.

O modelo de configuração apresentado por CONIC é análogo ao dos esquemas ilustrados nas Figuras (V.1 e V.2), para configuração estática e dinâmica respectivamente.

Para ilustrar as declarações da linguagem CONIC/C mostraremos o exemplo da enfermaria, utilizado também para apresentar a nossa linguagem de configuração no Capítulo VI, segundo a forma desenvolvida por KRAMER e MAGEE em [87].

No Capítulo V, seção V.4.5.1 foi mostrada a evolução da forma de estruturar hierarquicamente os sistemas em CONIC/C.

No exemplo que mostraremos, foram definidos dois tipos de configurações, uma para a cama do paciente e a outra para a enfermeira, declaradas simplesmente como módulos, e a configuração da enfermaria, composta por várias camas e uma enfermeira, foi declarada do tipo sistema ("system") por ser a de mais alto nível.

De forma resumida apresentamos a configuração da enfermaria e seus componentes na Figura (VII.19), para depois analisar as declarações da linguagem.

```
define patienttypes:maxbed,sensortype,alarmstype,
                    patientstatustype;
       const maxbed     = 16
       type sensortype = (bloodpressure,skinresistance,
                          temperature,pulse);
            alarmtype = (outofrange,sensorfault,noalarm);
            alarmstype = array[sensortype] of alarmtype;
            ......................
end.

module nurseunit;
   use patienttypes:alarmstype,patientstatustype,maxbed;
   entryport alarms[1..maxbed]:alarmstype;
   exitport query[1..maxbed]:signaltype reply patientstatustype;
   ..................
end.

module bedmonitor (scanrate:integer);
   use patienttypes:alarmstype,patientstatustype;
   exitport alarms:alarmstype;
   entryport status:signaltype reply patientstatustype;
   ..................
end.

system-ward;
   use bedmonitor;nurseunit;
   const nbed=4;
   create family k:[1..nbed]
            bed[k]:bedmonitor(100);
   create nurse:nurseunit;
   link family k:[1..nbed]
            bed[k].alarms to nurse.alarms[k];
            nurse.query[k] to bed[k].status;
end.
```

FIGURA VII.19 - CONFIGURAÇÃO DA ENFERMARIA

Podemos notar que é declarado, através do **define**, um arquivo contendo os tipos comuns, que são importados pelos diferentes módulos que os compartilham.

A declaração **use** define o contexto do conjunto de tipos de mensagens e de tipos de módulos necessários para o sistema.

Queremos salientar que os módulos de nível mais baixo, como nurseunit e bedmonitor no exemplo, declaram interfaces formadas pelas portas de entrada e saída locais a cada módulo.

As instâncias de tipos de módulos são criadas através da declaração **create**, algumas utilizando parâmetros, e podemos notar o uso do atributo **family** para criar um conjunto de módulos do mesmo tipo. O número que aparece entre parênteses depois do tipo do módulo indica o tamanho da pilha de memória necessária, expresso em palavras.

As conexões entre instâncias de módulos são realizadas através da declaração **link**, que liga portas de saída com portas de entrada de diferentes módulos, que precisam ter sido declaradas do mesmo tipo para poderem ser casados. Na declaração

de tipo da porta se definem o tipo da mensagem e o tipo de comunicação da porta, se é síncrona ou assíncrona. Aqui também é utilizado o atributo **family** para fazer um conjunto de ligações análogas.

As estruturas hierárquicas podem ser representadas no nível da configuração, aninhando especificações de configurações. No exemplo a configuração "system-ward" está formada a partir de outras duas configurações, "bedmonitor" e "nurseunit".

MAGEE em [104] apresenta também a declaração **include** que nomeia os módulos do sistema operacional e do sistema de comunicação a serem carregados em cada nó físico. No nível mais alto da hierarquia ele mostra também como é feito o **mapeamento físico** das estações através da indicação explícita do endereço físico do nó, na criação das subconfigurações. Por exemplo:

```
create  s1:keyin <1>
        s2:buff <2>
        s3:prout <3>
```

Já numa versão posterior, em [105] é apresentada uma linguagem de construção da configuração ("Configuration Language Builder") na qual é definido o mapeamento, explicitando para cada nó físico a lista de módulos que o compõem, por exemplo:

```
create at node (2,"t68k")
   TargExec;;
   temperature:sensor;
   Tscale:scale(Tfactor);
```

Nesta mesma versão, é colocado que os gerenciadores de recursos em CONIC não fazem parte do núcleo básico e são implementados por módulos do tipo tarefa separados.

Para apresentar a forma de implementar a configuração dinâmica precisaremos acompanhar a evolução das propostas.

Em [104] MAGEE apresenta as declarações utilizadas para definir as mudanças de uma configuração, que devem seguir à declaração:

change <nome da configuração>

Estas declarações são funções inversas das já apresentadas:

i) **unlink** de **link**

ii) **delete** de **create**

iii) **remove** de **use**

Vemos então que as mudanças são especificadas de forma separada e podemos ilustrar estes conceitos através dos dois exemplos de [87], onde o primeiro é uma extensão do exemplo da enfermaria, já apresentado, e o outro é uma modificação do sistema, substituindo um tipo de módulo por outro.

O primeiro exemplo, ilustrado pela Figura (VII.20), representa a **extensão da configuração** anterior na qual foi acrescentado um módulo que armazena dados para facilitar o sistema de monitoramento do paciente, chamado "datalogger". Depois da sua declaração, são especificadas as mudanças da configuração "ward", através da inclusão do tipo de módulo, a criação de uma instância dele, e a especificação de novas ligações.

```
module datalogger(recordfreq:integer);
   use patienttypes:patientstatustype,maxbed;
       loggertypes:patientdtype,replaydatatype;
   exitport getdata[1..maxbed]:signaltype
                        reply patientstatustype;
   entryport history:patientdtype
                        reply replaydatatype;
   [.....................................]
end.

change ward;
   use datalogger;
   create log:datalogger(500);
   link family k:[1..nbed]
          log.getdata[k] to bed[k].status;
end.
```

FIGURA VII.20 - CONFIGURAÇÃO ESTENDIDA DA ENFERMARIA

Esta extensão não modifica nada da configuração anterior, nem as conexões existentes dos módulos "bed" e "nurse"; portanto, se fosse feita dinamicamente, não seria necessário interromper a execução de nenhum dos módulos.

O próximo exemplo, pelo contrário, ilustra uma **modificação da configuração** original da Figura (VII.19), onde é substituido o módulo do tipo "nurseunit" por outro no qual a enfermeira possa exibir o histórico da informação armazenada pelo módulo acrescentado no exemplo anterior. Mostraremos primeiro na Figura (VII.21) as modificações decorrentes da substituição.

```
module enhancednurseunit;
   use patienttypes:alarmstype,patientstatustype,maxbed;
      loggertypes:patientdtype,replaydatatype;
   entryport alarms[1..maxbed]:alarmstype;
   exitport query[1..maxbed]:signaltype;
                        reply patientstatustype;
            history:patientidtype
                        reply replaydatatype;
   ................
end.

change ward;
   unlink family k:[1..nbed]
      bed[k].alarms from nurse.alarms[k];
      nurse.query[k] from bed[k].status;
   delete nurse;
   remove nurseunit;
   use enhancednurseunit;
   create newnurse:enhancednurseunit;
   link family k:[1..nbed]
          bed[k].alarms to newnurse.alarms[k]
          newnurse.query[k] to bed[k].status;
   link newnurse.history to log.history;
end.
```

FIGURA VII.21 - CONFIGURAÇÃO MODIFICADA DA ENFERMARIA

Neste caso, depois de declarar o módulo novo, são especificadas as mudanças, desfazendo as ligações anteriores, destruindo a instância e o tipo do módulo a ser substituido, incluindo o novo módulo, criando uma instância dele e fazendo as novas ligações.

Fazendo a inclusão e a substituição de uma vez só, supondo que já foram declarados os dois módulos novos, a configuração resultante seria especificada pela Figura (VII.22), na qual aparecem somente as mudanças.

```
system ward;
   use bedmonitor;enhancednurseunit;datalogger;
   const nbed=4
   create family k:[1..nbed]
          bed[k]:bedmonitor(100);
   create newnurse:enhancednurseunit;
          log  : datalogger(500);
   link family k:[1..nbed]
          bed[k].alarms to newnurse.alarms[k];
          newnurse.query[k] to bed[k].status;
          log.getdata[k] to bed[k].status;
   link newnurse.history to log.history;
end.
```

FIGURA VII.22 - CONFIGURAÇÃO ESTENDIDA E MODIFICADA DA ENFERMARIA

Os exemplos apresentados correspondem a um modelo centralizado de configuração dinâmica, no qual a unidade de substituição é o módulo de CONIC/C, que neste caso equivale a uma sub-configuração formada de módulos escritos em CONIC.

Como já mencionamos no Capítulo V, as idéias do projeto CONIC evoluiram em relação ao modelo de configuração dinâmica a ser implementado.

Por um lado surgiu a preocupação em relação à consistência do sistema durante a fase de reconfiguração. Para garantir a consistência do sistema, foram sugeridas duas soluções que eles colocaram de forma incipiente. Uma é a possibilidade de executar a especificação de mudanças de forma atômica, incluindo a lista de mudanças numa diretiva de tipo transação ("transaction") (MAGEE et alii [105]). A consistência estaria garantida pelas propriedades, já mencionadas no Capítulo V, da execução de uma ação atômica. A outra solução, apresentada em (MAGEE et alii [106]), é a inclusão, pelo próprio programador dos módulos escritos em CONIC, de condições de sincronização de maneira de não permitir nos trechos críticos do seu programa as interrupções, que possam ocorrer durante uma reconfiguração. Esta solução obriga o programador a estar ciente da possibilidade de ocorrer alguma reconfiguração que possa afetar seus módulos, e se prevenir contra possíveis interrupções, definindo condições de sincronização para garantir a consistência de cada módulo por ele programado. Isto, em ARGUS, é implementado na linguagem de programação, que para cada guardião permite definir variáveis estáveis, a partir das quais é retomado seu estado em caso de interrupção. Queremos recordar, como já foi mencionado em V.4.5.1, que na última versão de CONIC [106] foi mudada também a unidade de configuração, que agora é o nó lógico, conceito mais parecido com o subsistema de ARGUS, definido por BLOOM em [23].

Por outro lado, são colocadas em [106] algumas idéias sobre a possibilidade de implementar a configuração dinâmica de forma descentralizada, através de uma linguagem de configuração interativa, que pudesse ser utilizada em cada nó físico, e que permitisse até de combinar programas escritos em linguagens de programação em pequena escala diferentes, e não necessariamente só em CONIC. Como foi mencionado no Capítulo IV, na seção IV.1, existem outros projetos também evoluindo nesta direção.

COMPARAÇÃO DE CONIC COM A NOSSA PROPOSTA

O projeto CONIC, como já citamos anteriormente, foi um dos

trabalhos que norteou a nossa pesquisa, e em particular nos levou à decisão de definir uma linguagem separada de configuração, como consequência das vantagens que os autores apontaram.

Vamos apresentar algumas das diferenças da nossa proposta em relação ao projeto CONIC. A primeira que podemos citar é o fato deles terem partido de uma nova linguagem de LPP, definida como uma extensão de Pascal. Os autores acrescentaram, como mecanismo de comunicação e sincronização entre processos alocados em nós físicos iguais ou diferentes, a troca de mensagens síncrona e assíncrona, através de portas de entrada e saída locais, que precisam serem ligadas explicitamente. A troca de mensagens síncrona é parecida com o mecanismo de RPC.

As interfaces dos módulos de CONIC são definidas implicitamente através das declarações locais de portas de entrada e saída. As interfaces, desta forma, não estão separadas da parte de implementação como em Modula-2, e portanto não é possível a existência de diferentes módulos de implementação para uma mesma interface como no nosso caso. Esta característica nos parece dar uma flexibilidade e um poder bem maior à nossa proposta.

O contexto de uma sub-configuração em CONIC/C é definido pelas declarações **use** e **include** e não existem declarações de importação e exportação como no nosso caso, já que a comunicação é feita através de portas locais a cada módulo. Outra diferença reside no fato de que em CONIC/C não se identifica explicitamente em que estação lógica nem física encontra-se a instância do módulo ou da sub-configuração criada. Isto é realizado implicitamente ao se declarar o **create** no escopo da definição de uma sub-configuração. Somente no último nível existe, como já foi mencionado anteriormente, a identificação da estação física no **create**. A nossa especificação de estação lógica, complementada no final pela diretiva **LOAD**, nos parece bem mais clara para identificar a configuração lógica e física do sistema.

Somente nas últimas propostas de CONIC/C identificamos uma flexibilidade maior na capacidade de estruturação hierárquica do sistema, como já foi mencionado no Capítulo V.

Devemos reconhecer que em CONIC/C existe uma coerência maior na declaração **link**, que sempre liga explicitamente portas de entrada com portas de saída do mesmo nível hierárquico. Na nossa

proposta a declaração LINK liga interfaces de tipos de módulos com instâncias de módulos de implementação. Mas depois do primeiro nível da hierarquia, estas ligações ficam mais implícitas, porque, em lugar de serem nomeadas as interfaces de tipos de módulos, são nomeadas as sub-configurações às quais elas pertencem. O acompanhamento pode ser feito claramente através das definições dos componentes e das propagações das importações e exportações.

O projeto CONIC foi bem além do nosso, já que seus autores abordaram o problema da configuração dinâmica, enquanto o nosso só define a linguagem de configuração estática. Mesmo assim percebemos que os autores não chegaram ainda a uma versão final para o tratamento da configuração dinâmica, que, como já foi mostrado, tem vários problemas a serem tratados e resolvidos ainda. Este comentário se aplica também em relação ao tratamento de falhas, que está em desenvolvimento de forma separada, como foi mencionado em II.4.4.

VII.9 ALGUMAS CONSIDERAÇÕES FINAIS

Depois de ter comparado a nossa proposta com algumas linguagens e ambientes de programação distribuída, gostaríamos de levantar alguns pontos sobre os quais percebemos que não existe consenso ainda na literatura.

Temos apresentado no Capítulo V os requisitos necessários, tanto para a linguagem de LPP quanto para a linguagem de LPL, para montar um ambiente de programação distribuída. Nos exemplos analisados nem sempre esses requisitos todos são satisfeitos; na maioria dos casos só alguns deles foram considerados.

Uma primeira discussão a ser feita é em relação a ter uma única linguagem para montar o ambiente, ou duas separadas, uma para a programação dos módulos básicos, e outra distinta para a especificação da configuração estática e dinâmica. Como vimos, existem opiniões conflitantes sobre este tema. Por um lado, como já citamos no Capítulo II, MONTANARI em [114] defende o uso de uma única linguagem que estaria formada a partir de uma linguagem básica, por diferentes níveis de extensões satisfazendo os sucessivos requisitos, chegando a altos níveis de abstração, para a formulação completa do ambiente distribuído. Este foi

basicamente o enfoque seguido pelo projeto Cnet que desenvolveu o ambiente de programação distribuido, a partir da linguagem ADA, para uma arquitetura física composta de nós frouxamente ligados e dando suporte à configuração dinâmica. Nesta linha podemos citar os exemplos apresentados, tais como *MOD, SR e ARGUS, com a distinção de que *MOD dá suporte somente à configuração estática. Podemos deduzir, a partir destes exemplos, que não é imprecindível a separação das linguagens para implementar configuração dinâmica.

Tanto SR e ARGUS, como já foi apresentado, permitem a criação e destruição dinâmica de elementos básicos da linguagem, e as ligações podem ser feitas dinamicamente através do envio em mensagens de direitos de acesso ou de manuseadores, respectivamente. O componente que recebe estas informações poderá então se comunicar com o elemento ao qual pertencem estas informações. Com as ligações feitas desta maneira, a comunicação entre elementos da linguagem não precisa conhecer a sua alocação física, ou seja, a comunicação é independente da arquitetura física na qual executa o sistema. Mas por outro lado, existe uma dependência do sistema com a arquitetura física, pelo fato de que a função de criação de elementos básicos está ligada com a alocação física deles. Em ARGUS é enfatizada a vantagem de manter o controle de configuração no nível de programação do sistema, em relação à autonomia dos nós para decidirem a sua própria reconfiguração por razões técnicas ou políticas, por exemplo. Desta forma, está se mostrando a necessidade do processo de reconfiguração ser descentralizado.

Foi colocado enfaticamente pelo grupo de CONIC a importância para reconfiguração dinâmica e também para o processo de validação das mudanças do sistema, da separação das funções de criação, alocação e ligação de componentes básicos. ARGUS contorna em parte o problema de validação e consistência do sistema, através da implementação de guardiães, para os quais existe um estado estável a partir do qual é reiniciada a sua execução, seja depois de acontecer uma falha ou uma reconfiguração, e, por outro lado, pela inclusão de ações atômicas. Sobre este último ponto queremos colocar a opinião de ANDREWS [6] que não quis implementar ações atômicas a nível da linguagem SR, por considerar que a sua execução precisa de um

suporte substancial do sistema operacional. Dado que a linguagem SR está sendo desenvolvida para implementar sistemas operacionais, ele considerou que não correspondia incluir este conceito neste nível.

Dos modelos que apresentam linguagens separadas de configuração, como é o caso de C/MESA, Mascot3 e CONIC/C, podemos diferenciar cada caso. C/MESA, que não foi projetada para sistemas distribuidos, oferece somente configuração estática. Esta linguagem, por causa da separação total entre a definição das interfaces dos módulos e os tipos dos módulos de implementação correspondentes, oferece uma flexibilidade muito grande, para combinar diferentes módulos de maneiras distintas. Esta característica é oferecida em Mascot3 também, e em parte na nossa proposta. Mascot3 suporta configuração estática, e é mencionada em [151] a dinâmica também, embora não tenhamos conseguido bibliografia sobre isto.

Apesar da definição separada da linguagem de configuração, nem todas as funções são separadas e independentes. Tanto em Mascot3 quanto em MESA/C, as ligações são feitas implicitamente de forma parametrizada na criação de instâncias de componentes básicos. Esta forma de ligação dificulta em parte a reconfiguração dinâmica, já que para modificar alguma ligação é necessário destruir a instância, que em determinados casos, como por exemplo, num rebalanceamento ou realocação de componentes nos nós físicos, isto não seria preciso. CONIC/C apresenta uma separação total de funções e a reconfiguração dinâmica é expressa de forma separada, para permitir uma validação de forma mais simples. A maior crítica colocada a CONIC por DAY [46] é que só nas últimas versões seus autores enfrentaram o problema da preservação do estado do sistema e das condições de substituição. Adicionalmente, DAY comenta que a eliminação do controle a nível de programa da configuração ignora fatores que estão relacionados com a autonomia dos nós. Achamos que esta segunda crítica foi considerada na última versão de CONIC [106], na qual se delineam os passos futuros na direção de ter um controle maior em cada nó: primeiro, através da definição da unidade de configuração sendo o nó lógico, e segundo, propondo implementar um controle descentralizado, através de uma linguagem de configuração interativa a ser utilizada em cada nó.

Queremos salientar que as idéias, que acabamos de citar, coincidem em parte com as conclusões da tese de BLOOM [23], que propôs, como unidade de configuração para ARGUS, o subsistema, em lugar do guardião, e também indicou a importância de poder incluir o controle das condições de substituição, através de uma linguagem interativa. Na última versão de CONIC, é colocada a preocupação com as condições que devem ser satisfeitas no caso de reconfiguração dinâmica. Pelo fato de ter transferido a responsabilidade deste controle ao programador, que deve preservar os trechos críticos que não podem ser interrompidos, não consideramos que foi alterada a independência entre a LPP e LPL, já que estas condições não estão amarradas à configuração. Mas os autores reconhecem que esta solução não é elegante, e estão procurando achar soluções mais satisfatórias. Como já foi mencionado, eles levantaram a possibilidade de implementar transações para a execução das mudanças explicitadas de forma separada. Neste sentido, ARGUS garante uma consistência bem maior, por causa da forma de implementação do guardião já explicada anteriormente.

Queremos salientar que, de todas as propostas analisadas, somente ARGUS tratou mais aprofundadamente a garantia de consistência do sistema, tanto em relação à ocorrência de falhas, quanto em relação à configuração dinâmica. SR também considerou o problema de ocorrência de falhas, como já foi mencionado anteriormente, mas chegando à conclusão que a linguagem só devia se preocupar com falhas ao nível de hardware e não de software. Neste ponto também, eles consideraram que não é responsabilidade da linguagem tratar as falhas de software que possam acontecer. Esta pertence a outros níveis, como por exemplo o nível do programador do sistema ou do programa.

Podemos notar que praticamente todos os trabalhos utilizados para comparação implementam chamada remota de procedimento transparente, ou mecanismos equivalentes. Somente *MOD e SR oferecem outros mecanismos além do citado.

CAPÍTULO VIII

CONCLUSÕES

Este trabalho teve como objetivo o estudo de ambientes de programação distribuída, e em particular a definição e a implementação de um ambiente baseado numa linguagem de programação amplamente divulgada e aceita. Para isto foi necessário fazer uma pesquisa bibliográfica extensa, tanto em relação às características principais dos sistemas distribuídos, quanto em relação aos requisitos que a linguagem utilizada para implementar o ambiente precisaria satisfazer.

Pudemos distinguir dois níveis de programação, um referente à programação em pequena escala (LPP), isto é, para programar os componentes básicos do sistema, e o outro referente à programação em larga escala (LPL), no qual é definida a configuração do sistema, através da especificação dos componentes que o constituem e de suas ligações. Esta definição pode ser feita de forma estática ou dinâmica.

Estes níveis foram analisados separadamente, e depois de um estudo exaustivo, decidiu-se optar pela linguagem Modula-2, utilizada como base para o ambiente de programação distribuída a ser construído. Esta escolha baseou-se no fato da linguagem satisfazer vários dos requisitos necessários para uma linguagem de programação em pequena escala, e ainda oferecer facilidades para implementar as extensões requeridas para seu uso em sistemas distribuídos, sem modificar seu compilador.

A proposta apresentada inclui o mecanismo de comunicação e sincronização entre processos, denominado chamada remota de procedimento, cujas diferentes características e implementações foram analisadas detalhadamente, e que foi acrescentado a Modula-2, na sua forma transparente. Podemos afirmar que esta linguagem de programação estendida (LPP) satisfaz a maioria dos requisitos apontados, tais como:

i) independência de contexto
ii) interfaces bem definidas
iii) transparência de comunicação
iv) independência de interligação
v) independência de configuração

Foram escolhidas implementações simples, tanto para o núcleo de multiprogramação que dá suporte à RPC, quanto para a geração automática de "stubs", utilizados para implementá-la.

No segundo nível, depois da pesquisa bibliográfica sobre as características de configuração, foi definida uma outra linguagem a ser acrescentada a Modula-2, para a especificação da configuração estática de um sistema distribuido. Esta linguagem de programação em larga escala satisfaz os seguintes requisitos, considerados importantes nas discussões anteriormente apresentadas:

i) ela é declarativa
ii) possui uma unidade de configuração bem definida
iii) permite a definição clara de contexto
iv) possui funções totalmente separadas: por ex., definição de tipos, criação de instâncias, ligação de interfaces, alocação física
v) permite estruturação hierárquica
vi) inclui funções de mapeamento e carregamento

Foram apresentados vários exemplos mostrando, além das características da linguagem, o seu potencial.

Esta proposta foi comparada com várias outras escolhidas na literatura, como sendo as mais próximas a ela, e algumas tendo já sido utilizadas para desenvolver sistemas distribuidos específicos. Desta comparação pudemos extrair algumas conclusões interessantes. Por um lado, ficaram claras as vantagens que a nossa proposta apresenta e que já foram mencionadas. Por outro lado, percebemos as carências, que indicam os passos futuros a serem perseguidos para obter um ambiente de programação distribuida confiável, robusto e mais flexível.

Existem fundamentalmente duas áreas que não foram abordadas, apesar de uma delas ter sido estudada, levantando um conjunto de problemas para os quais não existem soluções únicas. Estamos nos referindo particularmente à configuração dinâmica. Em sistemas grandes, e especialmente em aplicações que não podem ser interrompidas, é da maior relevância permitir a introdução de mudanças de forma dinâmica, como já foi analisado nos capítulos anteriores. Infelizmente são muitos os problemas a serem

resolvidos, para poder implementar a configuração dinâmica garantindo a consistência do sistema. Existem algumas soluções neste sentido, como a implementação de ações atômicas para executar as mudanças, mas não existe um consenso em relação à inclusão desta solução ao nível da linguagem. Existe também a preocupação em relação à forma de expressar e garantir a validade de condições de substituição. Aqui também não está claro ainda, em que nível e de que maneira essas condições precisam ser implementadas. É discutido se é responsabilidade do programador especificá-las, ou, pelo contrário, se precisam existir mecanismos ao nível da linguagem para testá-las, de forma que fique transparente ao usuário, quando ele realizar uma substituição.

Gostaríamos de salientar que a definição e a implementação da nossa proposta foram concebidas, tendo em mente o objetivo posterior de estendê-la para dar suporte à configuração dinâmica, o que determinou várias das escolhas feitas.

A outra área que foi apenas mencionada é o tratamento de falhas e a confiabilidade. Esta área é realmente muito importante para garantir a confiabilidade de sistemas distribuídos, que se, por um lado, eles permitem implementar duplicação de funções e de parte do hardware para dar uma maior segurança ao sistema, por outro, pelo fato de depender de um conjunto de máquinas ligadas através de uma rede, eles possuem várias outras fontes de falhas que não existiriam num sistema de uma única máquina. Como vimos, são poucas as linguagens e as propostas que têm se preocupado com esta área, e ela necessita de pesquisas aprofundadas para encontrar soluções para seus problemas.

Queremos apontar que o ideal seria integrar as duas áreas mencionadas, já que na maioria dos casos o tratamento de falhas implica numa reconfiguração do sistema, e portanto deveria se achar soluções comuns.

Outro ponto, que gostaríamos de levantar sobre a evolução dos sistemas distribuídos, é em relação à flexibilidade necessária, por causa da combinação cada vez maior de componentes diferentes, tanto de software como de hardware. No caso de estar usando linguagens de programação distintas, é necessário que a linguagem de configuração e o mecanismo de comunicação e sincronização entre máquinas diferentes sejam compatíveis para

permitir a sua combinação. Foram vistos alguns exemplos nos capítulos anteriores em relação a esta situação.

A forma de processamento centralizada é mais adequada para a configuração estática. Entretanto, para satisfazer os requisitos de configuração dinâmica, vemos que cada vez mais se faz necessário um processamento descentralizado, de maneira a se ter uma autonomia maior em cada nó físico.

As propostas mais recentes levantam a necessidade de se ter um processamento interativo, sendo que cada nó físico deve conter as suas próprias informações de maneira a distribuir os dados, e a centralizar a menor quantidade de funções e de informações.

Os pontos levantados até aqui seriam os passos futuros a serem tratados para se obter um ambiente de programação distribuida flexível, robusto e confiável.

BIBLIOGRAFIA

[1] ABRAMSKY,S. & BORNAT,R. - **Pascal M: A language for Distributed Systems.** Queen Mary College, London, Research Report CLS 245, 1980.

[2] AHAMAD,M. & BERNSTEIN,A.J.-"An Application of Name-Based Addressing to Low-Level Distributed Algorithms". **IEEE Transactions on Software Engineering.** SE-11(1): 59-67, jan. 1985.

[3] ANDERSON,T. & LEE,P.A. - **Fault Tolerance, Principles and Practice.** Prentice Hall International, London, 1981, 369p.

[4] ANDREWS,G.R. - "The Distributed Programming Language SR - Mechanism, Design and Implementation". **Software-Practice and Experience**, 12(8): 719-753, ago.1982.

[5] ANDREWS,G.R. & SCHNEIDER,F.B. - "Concepts and Notations for Concurrent Programming". **ACM Computing Surveys.** 15(1): 3-43, mar. 1983.

[6] ANDREWS,G.R. & OLSSON,R.A.- **The Evolution of the SR Language.** Dept. of Computer Science, The University of Arizona, out. 1985, 26p. (Tech.Rep. 85-22).

[7] ANDREWS,G.R. & OLSSON,R.A. - **Report on the Distributed Programming Language SR.** Dept. of Computer Science, The University of Arizona, nov. 1985, 36p. (Tech.Rep. 85-23).

[8] ANDREWS,G.R. et alii - **The Design of the Saguaro Distributed Operating System.** Dept. of Computer Science, The University of Arizona, abr. 1985, (Tech.Rep. 85-9).

[9] APPELBE,W.F. - "A Survey of Systems Programming Languages: Concepts and Facilities". **Software-Practice and Experience**, 15(2): 169-190, fev.1983.

[10] BACARISSE,B. & WILBUR,S.- **Remote Procedure Call in Project Admiral**, Department of Computer Science, University College, London, 1985.

[11] BACARISSE,B.- **Remote Procedure Call - User Guide for RPC Release 2 and 3.** Department of Computer Science, University College, London, abr. 1986, 32p.

[12] BACON,J. -"An Approach to Distributed Software Systems". **ACM Operating Systems Review**, 15(4): 62-74, out. 1981.

[13] BALZER,R.M. -"Ports - A method for dynamic interprogram communication and job control". In: **Proc. AFIPS Spring Joint Computer Conf.**, vol 38 : 485-489, 1971.

[14] BANATRE,J.P., BANATRE,M.& PLOYETTE,F. - "The Concept of Multifunction: A General Structuring Tool for Distributed Operating Systems". In: **Proceedings of the 6th Conference on Distributed Computing Systems.** Cambridge, Mass. p.478-485, mai. 1986.

[15] BANINO,J. et alii - **CHORUS: An Architecture for Distributed Systems**, INRIA Research Report no. 42, nov. 1980.

[16] BATE,G. - "Mascot3: an informal introduction tutorial". **Software Engineering Journal**, 1(3):93-102, mai.1986.

[17] BERSHAD et alii- **A Remote Procedure Call Facility for Heterogeneus Computer Systems.** Tech. Rep. 86-09-10, Dept. of Computer Science, Univ. of Washington, set.1986.

[18] BIRRELL,A.D. et alii - "Grapevine: an exercise in distributed computing". **Comm. ACM**, 25(4): 260-274, abr. 1982.

[19] BIRRELL,A.D. & NELSON,B.J. - "Implementing Remote Procedure Calls". **ACM Transaction on Computer Systems**, 2(1): 39-59, fev. 1984.

[20] BIRRELL,A.D.-"Secure Communication Using Remote Procedure Calls". **ACM Transaction on Computer System**, 3(1): 1-14, fev. 1985.

[21] BIRRELL,A., LEVIN,R. & ROVNER,P. - **Current Operating System Work.** Systems Research Center, Digital Equipment

Corporation, 1985.

[22] BLACK,A.P. - **Exception handling: The case against.** Tech. Rep. 82-01-02, Department of Computer Science, The University of Washington, jan. 1982.

[23] BLOOM,T. - **Dynamic Module Replacement in a Distributed Programming System.** Ph.D. Thesis, Department of Electrical Engineering and Computer Science, MIT, Tech. Rep. MIT/LCS/TR-303, mar. 1983, 134p.

[24] BLOOM,T. - "Communication in the Common System". In: **Proceedings of the Workshop "Making Distributed Systems Work"**, Amsterdam, set. 1986.

[25] BOCHMANN,G.V. & RAYNAL,M. - "Structured Specification of Communicating Systems". **IEEE Transactions on Computers** C-32(2): 120-133, fev. 1983

[26] BRINCH HANSEN,P.- **Operating System Principles.** Englewood Cliffs, N.J., Prentice Hall, Inc., 1973, 366p.

[27] BRINCH HANSEN,P. - "The programming language Concurrent Pascal", **IEEE Transactions on Software Engineering**, SE-1,2: 199-206, jun. 1975.

[28] BRINCH HANSEN,P.- **The Architecture of Concurrent Programs.** Englewood Cliffs, N.J., Prentice Hall, Inc., 1977, 317p

[29] BRINCH HANSEN,P.- "Distributed Processes: A concurrent Programming Concept". **Communications of the ACM**, 21(11): 934-941, nov. 1978.

[30] BRINCH HANSEN,P. - "Edison: a Multiprocessor Language" - **Software-Practice and Experience**, 11(4):325-361, abr. 1981.

[31] BRON,C.- **Modules, Program Structures and the Structuring of Operating Systems.** Lectures Notes in Computer Science, No. 123, Springer-Verlag, 1981.

[32] BROWNBRIDGE,D.R., MARSHALL,L.F., & RANDELL,B. - "The Newcastle Connection - or UNIXES of the world unite!" **Software-Practice and Experience**, 12: 1147-1162, dez.

1982.

[33] BURGER,W.F. et alii - **Draft NIL Reference Manual**, Research Report RC 9732, IBM T.J., Watson Research Center, Yorktown Heights, NY, dez. 1982.

[34] CAMPBELL,R.M. & KOLSTAD,R.B. - "An Overview of Path Pascal's Design and Path Pascal User Manual". **SIGPLAN Notices**, 15(9): 13-24, set.1980.

[35] CASEY,L. - "Distributed Systems Already Work at BNR". In: **Proceedings of the Workshop "Making Distributed Systems Work"**, Amsterdam, set. 1986.

[36] CCITT, **Remote Operations. Model, Notation and Service Definition.** CCITT X.ros0 r ISO/DP 9072/1, Geneva, out.1986.

[37] CHERITON,D.R. et alii - "Thoth, a Portable Real Time Operating System". **Communications of the ACM**, 22(2): 105-115, fev. 1979.

[38] CHERITON,D.R. - "The V-Kernel: a software base for distributed systems". **IEEE Software**, 1(2):19-43, abr. 1984.

[39] CHERITON,D.R. & DEERING,S. - "Host Groups: A Multicast Extension for Datagram Internetworks". In: **Proceedings of the 9th Data Communications Symposium**, set.1985.

[40] Cnet - "Distributed Systems on Local Networks", Sottoprogetto P1". 2o. parte. Ed. ETS Pisa, jun 1985, 524p.

[41] COHEN,B., HARWOOD,W.T. & JACKSON,M.I. - **The Specification of Complex Systems.** Addison-Wesley Publ. Comp., Great Britain, 1986, 143p.

[42] COOK,R.P.-"*MOD - A Language for Distributed Programming". **IEEE Transactions on Software Engineering**, SE-6(6): 563-571, nov. 1980.

[43] COOPER,E.C. - "Replicated procedure call". In: **Proc. 3rd ACM Symposium on Principles of Distributed Computing**, Vancouver, BC, ago. 1984, 220-232.

[44] CUNHA,P.R.F., LUCENA,C.J.P. & MAIBAUM,T.S.E. -"On the Design and Specification of Message Oriented Programs". **International Journal of Computer and Information Sciences**, 9(3), jun. 1980.

[45] DA SILVA FILHO,N.A. - **Protocolo de chamada remota de procedimento**. Dissertação de Mestrado, Departamento de Informática, PUC/RJ, ago. 1987, 79p.

[46] DAY,M.S. - **Dynamic Configuration of Distributed Applications**, Laboratory for Computer Science, Massachussetts Institute of Technology, jul. 1985, 16p.

[47] DE REMER,F. & KRON,H.H.- "Programming-in-the-large Versus Programming-in-the-small". **IEEE Transactions on Software Engineering**. SE-2(2): 80-86, jun. 1976.

[48] DIJKSTRA,E.W. - "Guarded Commands, Non-determinacy and Formal Derivation of Programs". **Communications of the ACM**, 18(8):453-457, ago.1975.

[49] DIN (Deutsches Institut fur Normung), Programmiersprache, PEARL - Full PEARL, 1980.

[50] DOD,U.S. Department of Defence - **Requirements for ADA programming support environment** (Stoneman), 1980.

[51] DONAHUE,J. - "Integration Mechanisms in Cedar". **ACM SIGPLAN Notices**, 20(7): 245-251, jul. 1985.

[52] DULAY,N. et alii - **The CONIC Configuration language, version 1.3**. Imperial College Research Report DOC 84/20, nov. 1984, 12p.

[53] European Computer Manufacturer's Association, **ECMA Distributed Application Support Environment**, ECMA TC32-TG2/86/61, jul.1986.

[54] FANTECHI,A., INVERARDI,P. & LIJTMAER,N.- **The Cnet INTER-NODE communication mechanism**. Public. Int. del Progetto Finalizzato Informatica C.N.R. Progetto Cnet, 100, Pisa, Italia, dez. 1983, 26p.

[55] FELDMAN,J.A. -"High Level Programming for Distributed

Computing". **Communications of the ACM**, 22(6): 353-368, jun. 1979.

[56] FRANTA,W.R., JENSEN,E.D. & KAIN,R.Y. - "Real-Time distributed computer systems", **Advances in Computing**, vol. 20:39-82, 1981.

[57] GELERNTER,D. & BERNSTEIN,A.J. - "Distributed communication via global buffer". In: **Proc. Symp. Principles of Distributed Computing** (Ottawa, Canada, ago. 1982). ACM, New York, 1982, p. 10-18.

[58] GENTLEMAN,W.M. - "Message Passing Between Sequential Processes: the Reply Primitive and Administrator Concept". **Software-Practice and Experience**, 11(5): 435-466, mai. 1981.

[59] GHEZZI,C & JAZAYERI,M.- **Programming language concepts**. John Wiley & Sons, Inc., 1982, 327p.

[60] GIMSON, **Call Buffering Service**, Tech. Rep. No.19, Programming Research Group, Oxford University, 1985.

[61] GOOD,D.I. et alii- "Principles of proving concurrent programs in Gypsy". In: **Conf. Rec. 6th Ann. ACM Symp. on Principles of Programming Languages**, San Antonio, Texas, jan.1979, 10p.

[62] GRAY,T.E. - **Two Years of Network Transparency: Lessons learned from Locus**, The UCLA Computer Science Department Quarterly, Academic Programs, Spring Quarter 1985, Vol.13, No.2, 12p.

[63] HABERMAN et alii - **A Compendium of Gandalf Documentation**. Carnegie-Mellon University, Pittsburg, Pennsylvania, 1981.

[64] HAMILTON,K.G.- **A Remote Procedure Call System**. Ph.D. Thesis, Computer Laboratory, University of Cambridge, dez.1984, 109p. (Tech.Rep. No. 70).

[65] HERBERT,A.J. et alii - **The Mayflower Manifesto**. Computer Laboratory, Cambridge University, jun.1984.

[66] HOARE,C.A.R. - "Proof of correctness of data representations", **Acta Informatica**, vol.1:271-281, 1972.

[67] HOARE,C.A.R. - "Monitors: An operating system structuring concept". **Communications of the ACM**, 17(10): 549-557, out. 1974.

[68] HOARE,C.A.R. - "Communicating Sequential Processes". **Communications of the ACM**, 21(8): 666-677, ago.1978.

[69] HOLENDERSKI,L. - **An approach to mechanised verification of the behaviour of concurrent systems**, Tech. Rep., London, Imperial College, 1985.

[70] HOLT,R.C. et alii- **Structured Concurrent Programming with Operating Systems Applications**, Addison-Wesley, London, 1978, 262p.

[71] HOLT,R.C. - **Concurrent Euclid, the Unix System, and Tunis**. Addison-Wesley Publishing Company, 1983, 320p.

[72] HOPPE,J. - "A Simple Nucleus Written in Modula-2: A case study". **Software-Practice & Experience**, 10(9): 697-706, set.1980.

[73] ICHBIAH,J.D. et alii - "Rationale for the Design of the ADA Programming Language". **ACM SIGPLAN Notices**, 14(6): Part B, jun. 1979.

[74] INVERARDI,P., MAZZANTI,F., MONTANGERO,C. - **Configuration of Distributed Systems in ADA**, Nota Scientifica S-86-7, IEI-CNR, Pisa, Italia, 1986, 19p.

[75] JONES,A.K. et alii - "StarOs, a Multiprocessor Operating System for the Support of Task Forces". In: **Proceedings of the 7th Symposium on Operating Systems Principles**, California, p.117-127, dez. 1979.

[76] JONES,A.K. & SCHWARZ,P.- "Experience using Multiprocessor Systems - A Status Report". **ACM Computing Surveys**. 12(2): 121-165, jun. 1980.

[77] JOSEPH,M.- **Schemes for Communication**. Dept. of Computer Science, Tech. Rep. CMU-CS-81-122, Carnegie-Mellon

Univ., jun. 1981, 41p.

[78] KATAGUIRI,E.K. - "Alternativas para Comunicação entre Processos em Sistemas Distribuidos". **Boletim SCOPUS**. 6(67), jan. 1984, 12p.

[79] KERNIGHAN,B. & RITCHIE,D.- **The C Programming Language**. Prentice Hall, Inc., New Jersey, 1978.

[80] KERNIGHAN,D.W. & MASHEY,J.R. - "The Unix Programming Environment". **Software-Practice and Experience**, 9(1): 1-15, 1979.

[81] KIRNER,C. - **Desenvolvimento de suporte básico para sistemas operacionais distribuidos**. Tese de Doutorado, Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ, ago. 1986, 232p.

[82] KLINE,C. - "Complete vs. Partial Transparency in Locus". In: **Anais de European Unix Users Group**, Manchester, p.5-13, set.1986.

[83] KOCH,A.- **Menyma: Design and Implementation of a Message Oriented Language**. Master's Essay, Dept. of Computer Science, University of Waterloo, dez. 1980.

[84] KOCH,A. & MAIBAUM,T.S.E.- **MENYMA: A General Purpose Message Oriented Language**. Relatório Técnico, Departamento de Informática, Pontifícia Universidade Católica, Rio de Janeiro, RJ, 1981. 32p.

[85] KRAMER,J., MAGEE,J. & SLOMAN,M.- "Intertask Communication Primitives for Distributed Computer Control Systems". In: **Second Int. Conf. on Distributed Computing Systems**. Paris, abr. 1981.

[86] KRAMER,J. et alii - "CONIC: an integrated approach to distributed computer control systems". **IEE Proc.**, 130(1): 1-10, jan. 1983.

[87] KRAMER,J. & MAGEE,J. - "Dynamic COnfiguration for Distributed Systems". **IEEE Transactions on Software Engineering**, vol. SE-11(4): 424-435, abr. 1985.

[88] LAGES,N. dos S. - **Um gerador automático de "stubs" para a chamada remota de procedimentos num ambiente de programação distribuida baseado em Modula-2.** Tese de Mestrado em andamento, Programa de Sistemas, COPPE/UFRJ.

[89] LAMPSON,B.W. & REDELL,D.D.-"Experience with Processes and Monitors in Mesa". **Communications of the ACM**, 23(2): 105-117, fev.1980.

[90] LAUER,H.C. & NEEDHAM,R.M. -"On the Duality of Operating Systems Structures". In: **Proc. of 2nd Int. Symposium on Operating Systems.** IRIA, out. 1978, reimpresso: **ACM Operating System Review**, 13(2): 3-19, abr. 1979.

[91] LAUER,H.C. & SATTERTHWAITE,E.H. - "The Impact of MESA on System Design", **IEEE Ch1479-5/79/0000-0174**, p.174-182, 1979.

[92] LEACH,P.J. et alii -"The Architecture of an Integrated Local Network". **IEEE Journal on Selected Areas in Comm.**, nov.1983, p.842-856.

[93] LISKOV,B. et alii - "Abstractions Mechanism in CLU". **Communications of ACM**, 20(8): 564-576, ago.1977.

[94] LISKOV,B. - "Primitives for distributed computing". In: **Proceedings of the Seventh Symposium on Operating Systems Principles**, Pacific Grove California : 33-42, dez. 1979.

[95] LISKOV,B. et alii.- **CLU Reference Manual**. Lecture Notes in Computing Science 114, GOOS e HARTMANIS edit., Springer-Verlag, Berlin, 1981.

[96] LISKOV,B.- "Report on the Workshop on Fundamental Issues in Distributed Computing". **ACM Operating Systems Review**, 15(3): 9-38, jul. 1981.

[97] LISKOV,B. & SCHEIFLER,R. - "Guardians and actions: Linguistic support for robust, distributed programs". **ACM TOPLAS**, 5(2): 381-404, 1983.

[98] LISKOV,B. et alii.- **Preliminary Argus Reference Manual**. MIT

Programming Methodology Group Memo 39, out.1983.

[99] LISKOV,B.- **Overview of the ARGUS Language and System.** MIT Programming Methodology Group Memo 40, fev. 1984.

[100] LISKOV,B., HERLIHY,M. & GILBERT,L. - "Limitations of Remote Procedure Call and Static Process Structure for Distributed Computing". **Proc. 13th ACM Symp. on Principles of Programming Languages**, St. Petersburg, Florida, jan. 1986.

[101] LISTER,A.M. - "Multiprocess cooperation". In: **Proc. 8th Australian Computer Conference**, Canberra, ago. 1978.

[102] LISTER,A.M. et alii - **Distributed Process Control Systems: Programming and Configuration.** Research Report No.80/12. Dept. of Computing and Control, Imperial College, mai. 1980.

[103] LOQUES FILHO,O.G. & KRAMER,J. - **Transparent Module Replication in a Distributed System.** Department of Computing, Imperial College, London, ago. 1987.

[104] MAGEE,J.N. - **Provision of Flexibility in Distributed Systems.** Ph.D. Thesis, Department of Computing, Imperial College, London, abr. 1984, 135p.

[105] MAGEE,J., KRAMER,J. & SLOMAN,M. - **The CONIC Support Environment for Distributed Systems.** NATO Advanced Study Institute, Distributed Operating Systems: Theory and Practice, Izmir, Turkey, ago.1986, 19p.

[106] MAGEE,J., KRAMER,J. & SLOMAN,M. - **Constructing Distributed Systems in CONIC.** Imperial College, Department of Computing, Research Report DOC 87/4, Dept. of Computing, mar. 1987, 26p.

[107] MANNING,E., LIVESEY,N.J. & TOKUDA,H. - "Interprocess communication in Distributed Systems: One view". **Proc. IFIP**, 1980

[108] MAO,T.W. & YEH,R.T.- "Communication Port: A Language Concept for Concurrent Programming". **IEEE Transactions on Software Engineering**, SE-6(2): 194-204, mar. 1980.

[109] MARTELLI,M. & TARINI,F.- **Meccanismi di Comunicazione "Inter-Node" (Analisi e Proposte per una Lista di Requisiti).** Public. Int. del Progetto Finalizzato Informatica. C.N.R. - Progetto Cnet 15, Pisa, Italia, nov. 1980. 23p.

[110] Mc CARTHY,J. et alii - **Lisp 1 Programmer's Manual.** Computation Centre and Research Laboratory of Electronics, MIT Cambridge, 1960.

[111] MELLOR,P.V., DUBERY,J.M. & WHITEHEAD,D.G. - "Adapting Modula-2 for distributed systems". **Software Engineering Journal**, p.184-189, set. 1986.

[112] MILNER,R. - "A Calculus of Communicating Systems". In: **Lecture Notes in Computer Science**, vol. 92, New York, Springer-Verlag, 1980.

[113] MITCHELL,J.G. et alii - **Mesa Language Manual**, Report CSL-78-1, Xerox Parc, Palo Alto, California, 1978, 189p.

[114] MONTANARI,U. - "An Easy Interface for a Distributed System". **Workshop on "Models of Dialogue" - Theory and Application.** Linkoping, Sweden, jan. 1981, 11p.

[115] MULLENDER,S.J. & TANENBAUM,A.S. - "The Design of a Capability-Based Distributed Operating System", **Computer Journal** 29(4): 289-299, ago. 1986.

[116] NAUR,P.-"Revised Report on the algorithmic language Algol 60". **Communications of the ACM**, 6(1): 1-17, jan.1963.

[117] NEEDHAM,R.M. & WALKER,R.D.H.- "The Cambridge CAP Computer and its protection system". In: **Proceedings of the Sixth Symposium on Operating System Principles**, Purdue University, Lafayette, Indiana, nov. 1977.

[118] NEEDHAM,R.M. & HERBERT,A.J.- **The Cambridge Distributed Computing System.** Addison-Wesley, London, 1982.

[119] NELSON,B.J.- **Remote Procedure Call.** Ph.D. Thesis, Dept. of Computer Science, Carnegie-Mellon University, mai. 1981, 201p. (Tech.Rep.CMU-CS-81-119).

[120] NELSON,D.L. & LEACH,P.J. - "The Architecture and Applications of the Apollo Domain". **IEEE Computer Graphics & Applications**, abr. 1984, p.58-66.

[121] NOTKIN,D. et alii.-"Report on the ACM SIGOPS Workshop on Accommodating Heterogeneity". **ACM Operating Systems Review**, 20(2): 9-24, abr.1986.

[122] OUSTERHOUT,J.K., SCELZA,D.A. & SINDHU,P.S. - "Medusa: An experiment in distributed operating structure". **Communications of the ACM**, 23(2):92-104, fev. 1980.

[123] PANZIERI,F. & SHRIVASTAVA,S.K. - **Rajdoot: a Remote Procedure Call Mechanism Supporting Orphan Detection and Killing**. Technical Report Series 200, University of Newcastle upon Tyne, Newcastle upon Tyne, mai. 1985.

[124] PARNAS,D.L.- "On the criteria to be used in decomposing systems into modules". **Communications of the ACM**, 15(12): 1053-1058, dez. 1972.

[125] PARNAS,D.L. & SIEWIOREK,D.L. - "Use of the concept of transparency in the design of hierarchically structured systems". **Communications of the ACM**, 18(7): 401-408, jul. 1975.

[126] PARR, F.N. & STROM,R.E. - "NIL: A High-Level Language for Distributed Systems Programming". **IBM Systems Journal**, 22(1/2), 1983.

[127] PETERSON,J.L. - "Notes on a Workshop on Distributed Computing". **ACM Operating Systems Review**. 13(3): 18-27, jul 1979.

[128] PRIETO-DIAZ,R. & NEIGHBORS,J.M. - **Module Interconnection Languages: A Survey**. Department of Information and Computer Science, University of California Irvine, Irvine, ago. 1982, 69p.

[129] Proceedings of the Workshop "Making Distributed Systems Work", Amsterdam, set. 1986.

[130] REDELL,D.D. et alii - "Pilot: An Operating System for a Personal Computer". **Communications of the ACM**,

23(2):81-92, fev. 1980.

[131] REID,L.G. - **Control and Communication in Programmed Systems.** Ph.D. Thesis, Dept. of Computer Science, Carnegie-Mellon Univ. set. 1980. 148p (Tech. Rep. CMU-CS-80-142).

[132] RICHARDS,M. et alii - "Tripos - A Portable Operating System for Mini-computers". **Software-Practice and Experience**, 9(7): 513-526, jul. 1979.

[133] RICHARDS,M. & WHITBY-STREVENS,C. - **The BCPL language and its compiler.** Cambridge University Press, 1982, 173p.

[134] RITCHIE,D.M & THOMPSON,K. - "The Unix time-sharing system". **Communications of the ACM**, 17(7): 365-375, 1974.

[135] RODRIGUEZ,N. - **Um Sistema Operacional dedicado a Modula-2 para um microcomputador de 8 bits.** Dissertação de Mestrado, Departamento de Informática, PUC/RJ, abr. 1986.

[136] ROSSEM,P.H. van - **A network operating system kernel for remote invocations.** Department of Computer Science, Twente University of Technology, The Netherlands (MEMORANDUM NR.INF-82-14), dez.1982, 17p.

[137] RUGGIERO,W.V. & BRESSAN,G. -" Um modelo de programação para sistemas distribuidos". **Revista Brasileira de Computação.** 2(3): 131-149, set. 1982.

[138] SALTZER,J.H. - "Research Problems of Decentralized Systems with Largely Autonomous Nodes". **ACM Operating Systems Review.** 12(1): 43-52, jan. 1978.

[139] SCHLICHTING,R.D. & SCHNEIDER,F.B. - "Fail-stop processors: an approach to designing fault-tolerant computing systems". **ACM Transactions on Computer Systems**, 1(3): 222-238, ago. 1983.

[140] SCOTT,M.L.- "Messages vs. Remote Procedures is a False Dichotomy". **ACM Sigplan Notices**, 18(5): 57-61, mai. 1983.

[141] SEGRE,L.M. & KIRNER,C. - **Primitivas de Comunicação e**

Sincronização de Processos por Troca de Mensagens: Uma análise. Relatório Técnico, Departamento de Computação e Estatística, Univ. Federal de São Carlos, São Carlos, SP, dez. 1982, 41p.

[142] SEGRE,L. & STANTON,M. - "Análise de Programação Concorrente no Contexto de Sistemas Distribuidos". In: **Anais do II Congresso Latino-Iberoamericano de Investigacion Operativa e Ingenieria de Sistemas**, Buenos Aires, Argentina, ago. 1984, p.150-164.

[143] SEGRE,L. & STANTON,M. - "Modula-2: Suporte para o Desenvolvimento de Software Concorrente, In: **Anais do XVII Congresso Nacional de Informática**, nov. 1984.

[144] SEGRE,L. & STANTON,M. -"A Linguagem Modula-2 e seu ambiente de suporte de programação". In: **Anais do XVII Congresso Nacional de Informática**, nov. 1984.

[145] SEGRE,L.M. & STANTON,M.A. - "A construção de software distribuido usando Modula-2: Paralelismo, Comunicação e Confiabilidade". Artigo apresentado no **III Congresso Latino-Iberoamericano de Investigacion Operativa e Ingenieria de Sistemas**, Santiago de Chile, ago. 1986, 8p.

[146] SEGRE,L.M. & STANTON,M.A. - "A construção de software distribuido usando Modula-2: Linguagem de configuração". Artigo apresentado no **III Congresso Latino-Iberoamericano de Invertigacion Operativa e Ingenieria de Sistemas**, Santiago de Chile, ago. 1986, 8p.

[147] SHRIVASTAVA,S.K. & PANZIERI,F. - "The Design of a Reliable Remote Procedure Call Mechanism". **IEEE Trans. on Computers**, 31(7): 692-696, jul.1982.

[148] SHRIVASTAVA,S.K. - "On the Treatment of Orphans in a Distributed System". In: **Proceedings of the 3rd IEEE Symposium on Reliability in Distributed Software and Database Systems**, Florida, out.1983.

[149] SILBERSCHATZ,A. - "Port Directed Communication". **The**

Computer Journal, 24(1): 78-82, fev. 1981.

[150] SIMPSON,H.R. & JACKSON,K. - "Process synchronization in MASCOT", **The Computer Journal**, 20(4): 332-345, nov. 1979.

[151] SIMPSON,H.R. - "The Mascot method". **Software Engineering Journal**, 1(3): 103-120, mai. 1986.

[152] SLOMAN,M, MAGEE,J. & KRAMER,J. - "Building Flexible Distributed Systems in CONIC". In: **Proc. SERC Distributed Computing 84 Conf.**, University of Sussex, Brighton, set. 1984, 21p.

[153] SOLOMON,M.H. & FINKEL,R.A. - "The Roscoe Distributed Operating System". In: **ACM Sigops Proceedings of the 7th Symposium on Operating Systems Principles**, California, p.108-114, dez. 1979.

[154] SPECTOR,A.Z.- "Performing remote operations efficiently on a local computer network". **Communications of the ACM**, 25(4): 246-260, abr. 1982.

[155] STANTON,M. - "Dualidade e a Construção de Programas Concorrentes". In: **Anais do II Congresso da Sociedade Brasileira de Computação. IX SEMISH.** Ouro Preto, MG, SBC, jul. 1982, p.455-462.

[156] STANTON,M.A. et alii - "Uma experiência na montagem de um ambiente de suporte de programação em Modula-2". **Anais do XIII SEMISH e VI Congresso da Sociedade Brasileira de Computação**, Recife, jul. 1986, p.454-464.

[157] STANTON,M.A. et alii - "A Implementação de Modula-2 em microcomputadores de 8 bits". Artigo apresentado no **III Congresso Latino-Iberoamericano de Investigacion Operativa e Ingenieria de Sistemas**, Santiago de Chile, ago. 1986, 14p.

[158] STEVENS,W.P., MYERS,G.F. & CONSTANTINE,L.C. - "Structured Design", **IBM Systems Journal**, 13(2): 115-139, 1974.

[159] STROM,R. & YEMINI,S. - "The NIL distributed systems programming languages: A status report". **ACM SIGPLAN**

Notices, 20(5): 36-44, mai. 1985.

[160] STROUSTRUP,B. -"An Experiment with the Interchangeability of Processes and Monitors". **Software-Practice and Experience**, 12(11): 1011-1025, 1982.

[161] SVOBODOVA,L. - "Workshop Summary - Operating System in Computer Networks". **ACM Operating Systems Review**. 19(2): 6-39, abr 1985.

[162] SWINEHART,D.C., ZELLWEGER,P.T. & HAGMANN,R.B. - "The Structure of Cedar". **ACM SIGPLAN Notices**, 20(7): 230-244, jul. 1985.

[163] TANENBAUM,A.S. & RENESSE,R. van - "Distributed Operating Systems". **ACM Computing Surveys**. 17(4): 419-470, dez. 1985.

[164] TANENBAUM,A.S. & RENESSE,R. van - **A Critique of the Remote Procedure Call Paradigm**. Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands. Submetido ao SOSP 87, 11p.

[165] THEAKER,C.J. & FRANK,G.R. - "MUSS - A Portable Operating System". **Software-Practice and Experience**, 9(8): 633-643, ago. 1979.

[166] THEAKER,C.J. & FRANK,G.R. - "An Assessment of the MUSS Operating System". **Software-Practice and Experience**, 9(8): 657-670, ago. 1979.

[167] THOMAS,J.W. - **Module Interconnection in Programming Systems Supporting Abstraction**, Ph.D. Thesis, University of Utah, jun. 1976.

[168] TISATO,F. & ZICARI,R. - **Operating System and System Language Requirement List**. Public. Int. del Progetto Finalizzato Informatica C.N.R. - Progetto Cnet, 19, Pisa, Italia, dez. 1980, 19p.

[169] UNGER,C. (ed.) - **Command languages**. North Holland Publishing Company, Oxford, 1975, 401p.

[170] VETROMILLE,V.P. - **Um interpretador de uma linguagem de**

configuração para um ambiente de programação distribuido baseado em Modula-2. Tese de Mestrado em andamento, Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ.

[171] WALDEN,D.C.-"A System for Interprocess Communication in a Resource Sharing Computer Network". **Communications of the ACM**, 15(4):221-230, abr. 1972.

[172] WEITZMAN,C. - **Distributed Micro/Minicomputer Systems: Structure, Implementation and Application.** Englewood Cliffs, N.J., Prentice-Hall, Inc., 1980, 403p.

[173] WILBUR,S. & BACARISSE,B.- **Building Distributed Systems with Remote Procedure Call.** Dept. of Computer Science, University College London, 1986, 18p.

[174] WILLIAMSON,R. & HOROWITZ,E.-"Concurrent Communication and Synchronization Mechanisms". **Software-Practice and Experience**, 14(2): 135-151, fev. 1984.

[175] WIRTH,N. - "The Programming Language Pascal". **Acta Informatica**, 1: 35-63, 1971.

[176] WIRTH,N. - "Modula: A language for Modular Multiprogramming". **Software-Practice and Experience**, 7(1): 3-35, jan/fev. 1977.

[177] WIRTH,N. -"The Module: a system structuring facility in high-level programming languages". In: **Proceedings of the Symposium on Language Design and Programming Methodology.** Lectures Notes in Computer Science, no. 79, Springer Verlag, 1980.

[178] WIRTH,N. - **Modula-2.** Technical Report, Institut fur Informatik, ETH, Zurich, mar. 1980, 36p.

[179] WIRTH,N. - "The Personal Computer Lilith", In: A.I. Wassermann (ed.), **Software Development Environments**, IEEE Computer Society Press, 1981.

[180] WIRTH,N. - **Programming in Modula-2.** Berlin, Springer-Verlag, 1982, 176p.

[181] YAN,S; CHEN-CHAN,S. & SOL,M.- "An Approach to Distributed

Computing Software Design". **IEEE Transactions on Software Engineering**. 7(4): 427-435, jul 1981.

[182] YEMINI,S. & BERRY,D.M. - "A modular verifiable exception-handling mechanism". **ACM Transactions on Programming Languages**, 7(2): 214-243, abr. 1985.