TÍTULO: CONSTRUCTOR - UMA PROPOSTA DE UMA FERRAMENTA PARA PROJETO E IMPLEMENTAÇÃO DE SISTEMAS DISTRIBUÍDOS

Autor: Paulo F. W. Keglevich de Buzin

Instituição: Curso de Pós-Graduação em Ciência da Computação

RESUMO

No presente trabalho é apresentado o problema da programação distribuída e o relacionamento do ambiente de processamento distribuído com o ambiente de processamento paralelo, concorrente e seqüencial. Baseado no problema de processamento distribuído, é apresentado um modelo que distingue a parte física da parte lógica de um ambiente distribuído. É desenvolvido um modelo de arquitetura do sistema distribuído para a parte algorítmica, e baseado neste é definida a linguagem de alto nível. Constructor de programação distribuída com mecanismo de sincronização e de "schedulling" baseado em envio de mensagens.

A seguir, são apresentadas uma série de exemplos da eficácia da linguagem para resolver problemas importantes, que vão desde controle de periféricos em hardware até transações de Banco de Dados. Depois são apresentados aspectos de implementação. Finalmente, nas conclusões é constatada a introdução de um novo estilo de programação, inerente à programação em ambientes distribuídos.

SUMÁRIO

1. INTRODUÇÃO

Dentro do contexto do projeto da rede local da UFRGS - REDURGS [1], especial ênfase é dada ao software. O objetivo básico na abordagem do software é obter um suporte adequado para as várias aplicações possíveis sobre uma rede de processamento distribuído. A tarefa mais importante no desenvolvimento deste suporte é o projeto de uma linguagem de alto nível de programação distribuída onde seja oferecido ao projetista de sistemas mecanismos para sincronização e envio de mensagens sob forma de primitivas simples e poderosas, além de outras vantagens como veremos a seguir.

Os mecanismos e primitivas do sistema são projetados de tal forma que seja possível construir com simplicidade; topologias de rede, mecanismos de sincronização de processos, controle de tempo real, escalonamento de processos, etc... O nome escolhido para o sistema é CONSTRUCTOR porque foi projetado de modo a poder construir qualquer ambiente de processamento de programa.

O sistema CONSTRUCTOR é constituído de vários níveis de protocolos, como sugerido em [1]. Procurou-se diminuir a complexidade do kernel distribuído de modo a facilitar a sua implementação, mas sem limitar o poder de resolução de problemas na linguagem CONSTRUCTOR de alto nível.

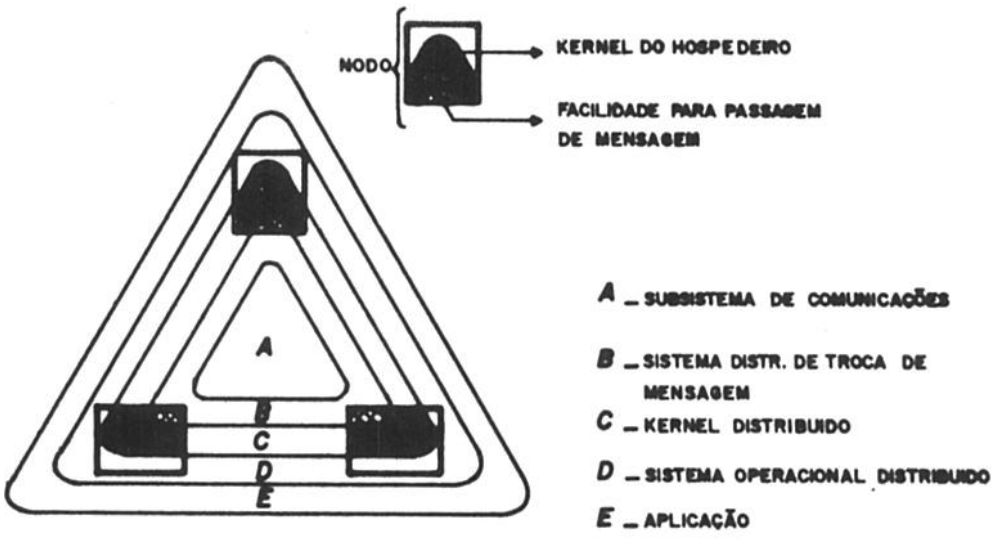

Fig. 1. Níveis de protocolo do ambiente distribuído CONSTRUCTOR.

2. O MODELO PARA O PROJETO DO CONSTRUCTOR

Ao resolver os problemas de processamento distribuído, também teremos soluções para problemas de ambientes de processamento paralelo, concorrente e seqüencial.

Neste trabalho entende-se como um ambiente de processamento paralelo aquele que possui vários processadores fortemente conectados para execução de processos, ao passo que no ambiente concorrente temos só um processador para a execução dos vários processos. Deste modo, um ambiente distribuído engloba um ambiente paralelo, que por sua vez engloba um ambiente concorrente, que ainda por sua vez engloba um ambiente sequencial (fig. 2).

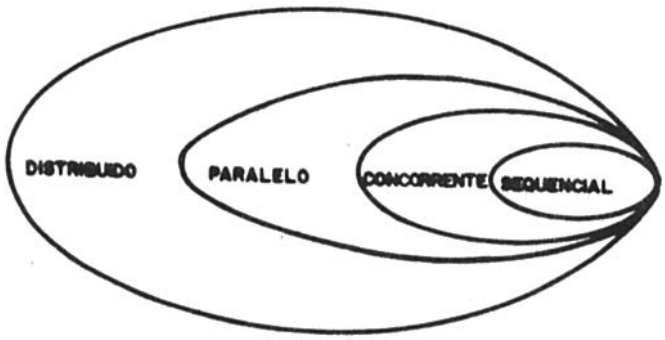

Fig. 2. Relacionamento entre os diferentes ambientes em computação.

Até ao nível de ambiente concorrente não existe preocupação com a localização de unidades de programa. Com o processamento paralelo existe a preocupação com a localização das unidades de programa ou com a associação algoritmo-processador para a computação de um programa, sem no entan

to preocupar-se com a característica do meio de comunicação. Já na construção de ambientes distribuídos existe a preocupação com o meio de comunicação, de modo que se procura alocar na rede módulos auto-contidos que se intercomunicam de modo que seja possível controlar as situações de erro e de retardo de mensagens.

O processamento distribuído necessita de dois níveis de esquema de programação para obter uma abstração da distribuição [3], ou seja, a especificação da configuração deve ser independente da implementação das partes algoritmicas do programa, e vice-versa. Desta maneira o sistema apresentaria a característica de poder construir de uma maneira versátil um ambiente distribuído.

Dentro do modelo escolhido para o ambiente do sistema CONSTRUCTOR, é possível distinguir entre o sistema real (redes, nodos, hospedeiros, processadores e memórias) e o sistema lógico ou parte algoritmica (processos e módulos), os quais depois de definidos podem ser associadas para construir um sistema como um todo.

A concepção da parte algoritmica do constructor parte do modelo de arquitetura distribuído (Distributed System Architecture Model [2]). Dentro deste modelo, um dos maiores objetivos no projeto de um sistema operacional distribuído é fornecer aos usuários objetos abstratos implementados por SERVERs. Um objeto abstrato pode ser especificado por: um conjunto de estruturas de dados e um conjunto de operações ou funções. Um SERVER pode ser construído por outros SERVERs, de forma que é possível o estabelecimento de uma hierarquia.

Incluiu-se no modelo o conceito de CLIENT (baseado no conceito de procedimentos nos "Data Flow Description" da análise estruturada) que atuam sobre os SERVERs para, por meio de suas operações, implementarem aplicações ao usuário.

Os módulos podem ser CLIENTs, SERVERs e INTERFACEs. Os únicos módulos que implementam operações são os SERVERs. Os processos dentro de um CLIENT só podem comunicar-se entre si por meio de operações implementadas em SERVERs. O módulo INTERFACE representa a interconexão entre dois ambientes diferentes, permitindo deste modo incluir na configuração de um ambiente elementos inteligentes tais como terminais, periféricos em geral, interfaces com outras redes de comunicações, transdutores ou coletores de processos de fabricação, etc... Também permite representar a interconexão entre ambientes diferentes de software (o que interessa para sistemas operacionais). Este tipo de módulo interage com o seu ambiente também por meio de operações em SERVERS, e somente os SERVERs podem atuar sobre as interfaces. Um interface é composto de um protocolo, um conjunto de dados em uma dada formatação e um conjunto de funções de atuação sobre o mesmo. Um SERVER acoplado a um interface pode funcionar como "driver" em software e implementar recursos para o ambiente.

O meio de comunicação entre os elementos do sistema real é assumido como fracamenta conectado ("losely conected") possuindo um retardo de modo que a produção de um evento e a sua materialização é distinguível. No caso do meio de comunicação ser o elemento passivo MEMÓRIA (compartilhado), então a comunicação é fortemente conectada e a produção de um evento e a sua materialização é indistinguível, pois não há retardo na comunicação. A comunicação sempre se dá entre elementos ativos, a maneira pela qual ocorre a comunicação é definida pelo protocolo, entre os mesmos [13].

De acordo com o modelo considerado, cada camada de protocolo da rede interage horizontalmente por meio de CLIENTs e SERVERs, e verticalmente por meio de INTERFACEs (fig. 2).

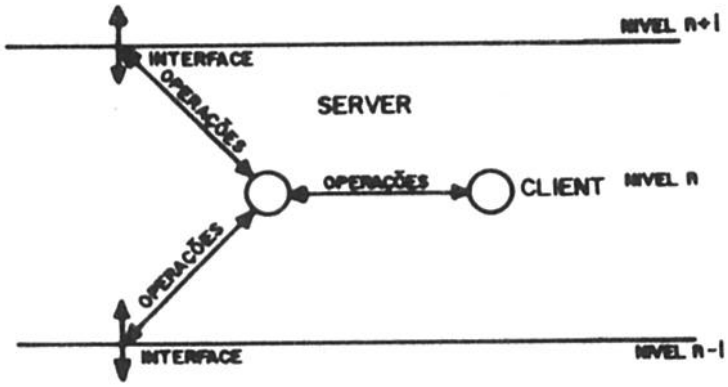

Fig. 3. Interação entre os componentes de um ambiente.

Desta forma a comunicação horizontal seria dentro do ambiente do sistema e a comunicação vertical seria para fora do ambiente do sistema. Assim, a interconexão entre dois ambientes se daria por meio do INTERFACE. Com este modelo é possível implementar a intercomunicação entre dois hospedeiros via SERVERs. (fig. 3).

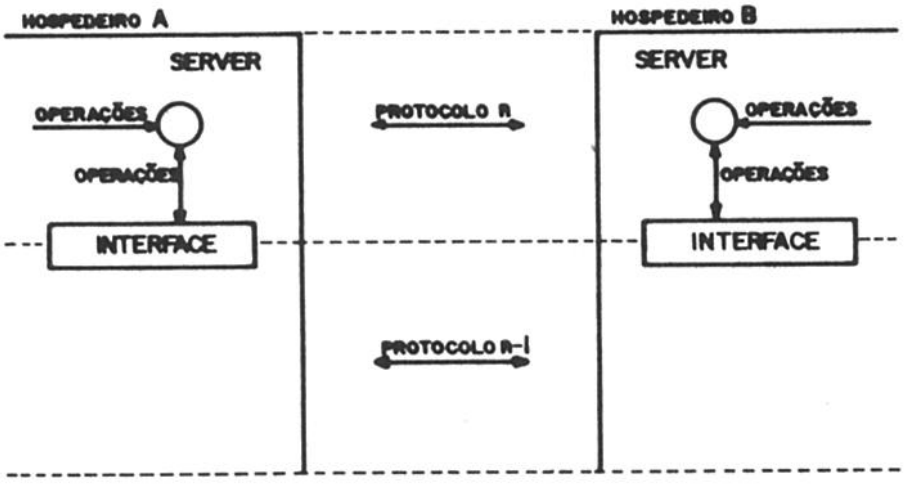

Fig. 4. Interconexão entre ambientes em dois níveis de protocolo.

Entende-se como ambiente o conjunto de recursos disponíveis e acessíveis de uma maneira uniforme dentro de um sistema único por meio de uma série de serviços fornecidos. Assim por exemplo, um nível de protocolo em um modo de rede pode ser considerado como ambiente. Da mesma forma, os elementos de um controlador de periféricos ou de processo podem ser considerados como um ambiente.

Um ambiente é construído definindo os elementos reais que o constituem e as conexões entre os mesmos, a seguir são definidos os elementos lógicos (CLIENTS, SERVERS e INTERFACES) que depois são localizados sobre os elementos reais.

O sistema CONSTRUCTOR é definido de forma que cada elemento ativo necessite apenas ter os endereços dos outros elementos do mesmo ambiente e suas condições de acesso. O restante das informações necessárias para verificação de disponibilidades de cada elemento é de responsabilidade única do próprio elemento ou do elemento ativo mais fortemente conectado ao mesmo. Isto facilita a manutenção da consistência em um sistema e reduz drasticamente o acúmulo de informações sobre o ambiente em cada elemento.

Por operações entende-se um conjunto de funções ou instruções executadas de modo exclusivo causando transformações sobre dados e/ou estados do sistema. As execuções de operação são mutuamente exclusivas e podem ser feitas por qualquer processo, seja CLIENT ou SERVER, desde que tenha acesso às mesmas. A execução de uma operação pode ser feita por envio de informação (comunicação por mensagens) ou por envio e recepção de informação (chamada remota de funções ou procedimentos). No último caso, o processo que ativou a operação fica suspenso até a completa execução da mesma.

A seguir é apresentada a proposta da linguagem CONSTRUCTOR, de maneira suscinta e informal. Esta primeira proposta ainda não é definitiva e está em processo de veri-

ficação de consistência e de potencialidades. As considerações de projeto que levaram a escolha desta alternativa como proposta de linguagem, assim como os detalhes de especificação, estão no trabalho de Tese presentemente em desenvolvimento.

3. A PROPOSTA

A linguagem CONSTRUCTOR segue o modelo previamente definido, possuindo, portanto, quatro tipos de módulos: CLIENTs, SERVERs, INTERFACE HARDWARE e INTERFACE SOFTWARE. As operações possíveis são definidas somente nos processos dos módulos tipo SERVER.

Processos podem se comunicar de duas maneiras diferentes: diretamente, por troca de mensagens ou indiretamente, lendo e escrevendo variáveis compartilhadas. Processos também podem se sincronizar de duas maneiras básicas: diretamente, por sinalização explícita, ou indiretamente, por teste e alteração de variáveis compartilhadas.

Os conceitos de SERVERs e operações permitem implementar com simplicidade qualquer dos modos de comunicação e sincronização acima descritos.

O conceito de módulos da linguagem CONSTRUCTOR é uma extensão da idéia de Wirth [4], onde os módulos isolavam partes do programa dependentes do tempo e de sincronização. No caso do CONSTRUCTOR, os módulos também isolam características funcionais e sintáticas diferentes. Isto concedeu maior clareza e concisão à linguagem, permitindo inclusive, a detecção de vários erros lógicos (inclusive de programação distribuída) em tempo de compilação.

Dentro dos módulos CLIENTs e SERVERs são definidos qualquer número de processos. Os módulos INTERFACE serão de dois tipos: Hardware e Software. O tipo Hardware é voltado principalmente para controle de hardware e de tempo real. O tipo SOFTWARE é voltado para a intercomunicação de programas em software.

Os módulos podem ser compilados separadamente e ligados durante a inicialização e carga. Neste caso apenas é necessário indicar quais as operações que fornecem para e

utilizam do ambiente em que farão parte, assim como os respectivos parâmetros.

3.1. Tipos e declarações

As declarações são similares as do Pascal [11] com algumas diferenças. Os tipos pré-definidos são booleano (BOLL), inteiro (INT), real (REAL), caracter (CHAR) e palavra (WORD). É possível ao usuário definir os seus próprios tipos de variável usando RANGE para definição de intervalos, ENUM para definição de escalares, REC para definição de registros, CAP para definição de "capability" e SET para definição de conjuntos. De uma forma geral tem-se:

```
TYPE <nome> = <def-tipo>;
onde <def-tipo> pode ser:
RANGE (<intervalo-de-escalar>)
ENUM  <lista-de-escalar>)
REC   (<lista-dos-campos>)
CAP   (<lista-de-operações>)
SET   (<designação>), onde <designação> pode ser
                      INT, CHAR, WORD ou ENUM.
```

Ao usuário é permitido declarar somente estes cinco tipos. Os tipos RANGE, ENUM e REC são idênticos aos do PASCAL [11] com a sintaxe do EDISON [9]. O que é diferente é o CAP, que é similar ao registro (REC) só que ao invés de campos, define uma lista de entradas as quais podem ser conectadas a operações em módulos por comandos de atribuição. Esta atribuição é válida quando os parâmetros são compatíveis. O tipo SET é o mesmo do PASCAL, porém com sintaxe diferente na declaração.

A declaração de variáveis introduz um ou mais identificadores e define os seus tipos, tal como o PASCAL. A diferença é a declaração de arranjos que é feita como no exemplo a seguir:

VAR linha [1..10]: CHAR
declara um arranjo de 10 caracteres.

No caso de parâmetros de procedimentos e operações é possível deixar em aberto a especificação de intervalo do arranjo, especificando com um '*' como em arranjo [*,*]:INT. Internamente Arranjo é visto como Arranjo [1:n, 1:m]:INT com n e m fixados na chamada do procedimento.

A declaração de PROCEDURE e FUNCTION é idêntica ao PASCAL [11], com as mesmas regras, com a diferença de ser possível passar variáveis do tipo RANGE ou INIT como parâmetro e usá-las para declarar variáveis do corpo do procedimento ou função. Isto permite a expansão dinâmica de variáveis na pilha de um processo sem os problemas de administração de memória do POINTER em PASCAL.

As constantes são declaradas de maneira idêntica ao PASCAL. As declarações de tipos e constantes devem ocorrer no início do programa, de módulos ou de processos.

O tipo pré-definido WORD corresponde e uma palavra da máquina onde reside o sistema e é de tamanho equivalente a INT ou BOOL. Com este tipo é possível realizar diretamente operações lógicas ou de aritmética inteira sem sinal e executar código previamente carregado por meio de um interface.

As variáveis só podem ser declaradas dentro de SERVERS e/ou processos.

3.2. Mecanismo de sincronização

O conceito de SERVER é praticamente idêntico ao de RESOURCE [5,8] com algumas pequenas variantes nas construções sintáticas.

As operações e os comandos de ativação são primitivas e se assemelham ao conceito de primitivas de entrada e saída para sincronização de processos de Hoare [6], com a diferença que o processo receptor da mensagem não precisa saber a identidade de quem envia a mensagem. As operações são uma generalização das "procedures" e de passagem de mensagem.

Estas operações são definidas por comandos SELECT os quais são baseados nos comandos guardados de Dijkstra [7], e são ativados por comandos SEND, para envio de mensagens, ou por comandos CALL, para chamada "procedures" onde o processo chamador fica suspenso até que conclua a operação chamada. Estas primitivas são as mesmas apresentadas por Andrews [5].

Os comandos de entrada declaram, sincronizam e sequenciam operações. Cada comando de operação tem a forma:

SELECT <operação> {ALSO <operação>}[ELSE <comando>]
ENDSEL;

<operação>::=<nome-de-operação>[<parâmetros-formais>)][WHEN <expressão-booleana>] [BY <expressão aritmética>] DO <lista-de comandos>.

A guarda da operação consiste no nome da operação e na expressão booleana. O nome da operação habilita a guarda se existe pelo menos uma invocação pendente. Se existem mais de uma invocação pendente para a qual a expressão booleana seja verdadeira, as invocações serão ordenadas pela expressão aritmética, de acordo com o menor valor (se a

opção BY é omitida, a ordenação é a de tempo de ativação).

O comando SELECT tem a seguinte semântica [8]: se pelo menos uma guarda de operação é verdadeira, uma é selecionada não-determinísticamente; se existe mais de uma invocação da operação selecionada que satisfaça a expressão de sincronização, aquela que minimiza a expressão de seqüencialização é escolhida; se nenhuma guarda é verdadeira e ELSE é presente, então este é selecionado; se nenhum comando de operação pode ser selecionado, o comando SELECT é retardado até que uma seleção seja possível. (Um comando SELECT com uma guarda ELSE nunca retarda). Uma vez que a seleção é feita, os correspondentes comandos são executados com os parâmetros reais da invocação escolhida. O comando SELECT termina assim que terminam os comandos selecionados para execução.

3.3. Comandos

A linguagem CONSTRUCTOR contém quatro tipos de comandos: seqüencial (nulo, aborto de programa, atribuição e chamada de procedure); alternação, iteração e comandos para a invocação e execução de operações.

Os comandos seqüenciais são os seguintes:

nulo:	SKIP
aborta programa:	ABORT
atribuição:	<lista-de-variáveis>:=<lista-de-expressões>
alternação:	IF <expressão-booleana> DO <comando> {ALSO <expressão-booleana> DO <comando>]} ENDIF
iteração:	WHILE <exp-bool> DO <comandos> {[ALSO <exp-bool> DO <comandos>]} ENDWHILE
saída de iteração:	OUT
chamada de procedure:	<nome-de-proc>[<parâmetros-formais>)]

Nos comandos de alternação e iteração é possível usar, no fim, um comando guardado [ELSE <comandos>] que é por definição a negação dos outros comandos guardados.

Para a concorrência a comunicação temos os seguintes comandos:

operação: SELECT <corpo da operação> ENDSEL
chamada: CALL <designação-de-operação>[(parâmetros-reais)]
envio: SEND <designação-de-operação>[(parâmetros-reais)]
concorrente: CONC <lista-de-chamadas-ou-envio-de-operações> ENDCONC

A designação de operação é o nome de uma operação definida por um SERVER ou um campo de uma variável tipo CAP. Os nomes de tais designações são subscritos se o processo, SERVER ou variável tipo CAP, foram declarados como um arrays.

O comando SELECT já foi descrito na secção anterior. O comando de chamada termina quando a operação designada foi executada e os resultados, quaisquer que sejam, retornam. O comando envia termina tão logo os parâmetros reais foram salvos para transmissão; conseqüentemente nenhum resultado retornará se houverem parâmetros resultantes. Um comando concorrente CONC contém um ou mais comandos de chamada ou envio de mensagens que são invocados em paralelo, pelo menos conceitualmente. Este comando termina quando todos os comandos da lista terminarem. O critério de passagem de parâmetros pode ser por valor (by-value), por referência VAR (by-reference) e por resultado RES (by-result). Uma exceção ocorre quando o parâmetro for por referência (VAR) e o meio de intercomunicação entre o processo que ativa o comando de comunicação e o processo que implementa a operação for fracamente conectado (não há compartilhamento de memória). Neste caso, o efeito será como se o parâmetro fosse por resultado (RES).

3.4. Processos

Um processo contém um conjunto de variáveis, procedimentos, funções e comandos. Tem a forma:

```
PROCESS <nome> ["["intervalo"]"] [AREA = <tamanho-area>];
[declaração de variáveis]
[declaração de procedimentos e funções]
lista de comandos
ENDPROS <nome>;
```

Um processo executa um comando por vez e termina quando a sua lista de comandos termina. As variáveis declaradas dentro de um processo são locais a ele. Quando estiverem em SERVERs os processos podem implementar operações. As operações são automaticamente disponíveis a outros processos em um mesmo SERVER. Os processos e os comandos SELECT permitem a programação de operações de exclusão mútua. Os processos são o único mecanismo para exclusão mútua.

A prioridade dos processos na criação é, por definição, mínima, podendo ser obtida ou alterada por meio da variável PRIORITY associada a cada processo.

Se um intervalo é especificado no cabeçalho de uma declaração de processo, uma família de processos idênticos é criada, uma para cada valor do intervalo. Dentro de uma instância, a variável especial MYPROCESS contém a identidade do processo.

As variáveis de um processo são temporárias e existem enquanto existir o processo. É admitida recursividade dentro dos processos.

O termo AREA define a área de trabalho do processo (pilha), onde o mesmo deve colocar as variáveis dos seus procedimentos e funções.

Adicionalmente, é possível declarar um procedimento tipo biblioteca dentro de um processo. Desta forma, quando o procedimento for chamado, o respectivo código será procurado na biblioteca, carregado na pilha do processo e posto em execução. Esta também é uma elegante maneira de obter "overlay" sobre a área de dados do processo. A forma geral da declaração é a seguinte [9,10]:

LIB PROCEDURE <nome>[(<parâmetros formais>)]

3.5. Construção de ambientes

Em programação distribuída a especificação e construção de ambientes adquire especial importância [3]. Depois de especificados os módulos e interfaces, surge o problema de onde localizá-los e de como fazer a comunicação entre os mesmos, se direta (memória compartilhada) ou indireta (envio de mensagens). Para resolver este problema a linguagem deve permitir a descrição dos elementos de um sistema físico onde irá operar o algoritmo programado, e a descrição do tipo de interconexão que existe entre os mesmos. Ao definir a interconexão fica definida a topologia e o modo de comunicação entre os elementos. Finalmente, sobre os elementos são localizados os módulos.

No sistema CONSTRUCTOR foram considerados quatro tipos de ambientes físicos. O primeiro é o de uma rede interconectando vários hospedeiros, o segundo é o de um hospedeiro contendo mais de um processador, que podem compartilhar uma memória, o terceiro é o de um hospedeiro com um processador, e o quarto é uma biblioteca de armazenamento de programas (módulos ou procedures), para futura e ligação e uso em outros ambientes.

A estrutura geral para construção dos ambientes é a seguinte:

A. Para redes (ambientes de processamento distribuído)

```
CONSTRUCT <nome-arquivo> = NET[(<endereço-de-nodo>)]
          ELEMENT {<nome-N> = NODE (<ender>);}
          CONECT {(<nome-N>,<nome-N>);}
          LOCALIZE {(<nome-módulo>,<nome-N>;}
```

B. Para Hospedeiros com mais de um processador (para ambientes de processamento paralelo):

```
CONSTRUCT <nome-arquivo> = HOST[(<endereço-de-nodo>)]
          ELEMENT {<nome-P> = PROCESSOR (<ender>);}
                  {<nome-M> = SHAREDMEM (<ender>);}
          CONECT {(<nome-P>,<nome-P>);}
                 {(<nome-M>,<nome-P>);}
          LOCALIZE {(<nome-modpross>,<nome-P>)}
                   {(<nome-modpross>,<nome-M>)}
```

C. Para hospedeiro com um processador (para ambientes de processamento concorrente).

```
CONSTRUCT <nome-arquivo> = HOST [,(<endereço-de-nodo>)]
```

D. Para biblioteca:
para procedimentos:

```
CONSTRUCT <nome-arquivo> = LIBP;
<declaração-de-tipos-e-constantes>
PROCEDURE <definição-formal>
<corpo-do-procedimento>
```

ou para módulos:

```
CONSTRUCT <nome-arquivo> = LIBM;
<declaração-de-tipos-e-constantes>
<declaração-de-módulos>
```

Nos casos A e B todos os elementos do ambiente no nível da declaração CONSTRUCT devem estar localizados pela declaração LOCALIZE, não importando se foi declarado CONSTRUCT dentro de um módulo ou em todo o ambiente. Já o mesmo não ocorre com as opções C e D.

A unidade léxica CONSTRUCT é o símbolo inicial da gramática da linguagem. O termo <endereço-de-nodo> é o endereço de um nodo dentro de uma rede; quando é emitido, é assumido como sendo nodo onde é carregado o programa. No caso de uma rede, o termo <endereço-de-nodo> pode ser também o ponto de conexão entre duas redes.

O termo <ender> pode ser endereço de nodo, número de processador ou endereço de início de memória. O termo <nome-módulo> pode ser nome de INTERFACE, SERVER ou CLIENT. O termo <nome-modpross> pode ser nome de módulos ou de processos.

No caso de biblioteca, é construído um arquivo em condições de ser ligado em tempo de carga de programa (LIBM) ou durante a execução de programa (LIBP).

A declaração CONSTRUCT só pode ocorrer no início do programa ou no início de módulo CLIENT ou SERVER.

Deste modo, estes módulos podem conter expansões do ambiente em que estão contidos (por exemplo: sub-redes ou um multiprocessador).

3.6. Servers

Conforme o modelo, os SERVERs definem um conjunto de operações e encapsulam um conjunto de dados permanentes sobre os quais atuam as operações. A estrutura básica é similar e dos Recursos (RESOURCE [5,8]) com algumas variantes adicionais. Um SERVER contém no mínimo um processo e,

opcionalmente, variáveis permanentes, ou procedimentos compartilhados, ou outros SERVERs e CLIENTs. Os processos, as variáveis permanentes e os procedimentos compartilhados do SERVER são considerados fortemente conectados e as variáveis permanentes possuem uma granularidade de uma palavra, de modo que o acesso a um elemento é uma transação atômica, mutuamente exclusiva. Os processos de um SERVER só podem acessar as variáveis permanentes do seu próprio módulo. Os SERVERs também podem utilizar operações definidas por outros módulos no mesmo nível, ou do módulo imediatamente superior hierarquicamente, ou dos módulos imediatamente inferiores.

Os comandos de inicialização de um SERVER, se existirem, devem ser comandos seqüenciais colocados no início do módulo que somente acessem as variáveis permanentes do módulo.

De um modo geral a forma sintática é a seguinte:

```
SERVER <nome-sv>["["<intervalo>"]"];
[<declaração-de-variáveis-e-tipos>]
{OP<nome-operação>[(<parâmetros-formais>)]["["
                        <modo-de-invocação>"]"]}
DEFINE {<nome-operação>["["<modo-de-invocação>"]"]}
UTILIZE {<nome-operação>[(<parâmetros-formais>)]
                  [IN <nome-MOD>][WHEN <exceção>]}
[SHARED
 <declaração-de-procedimentos-compartilhados>
 ENDSHARED;]
[INICIALIZE
 <comandos-de-inicialização>
 ENDINIC;]
[<construção-de-ambiente>]
{<declaração-de-processos>}
ENDSERVER;
```

No caso de ser um SERVER já compilado em arquivo então tem-se:

```
SERVER <nome-su>["["intervalo "]"]
[<declaração-de-tipos>]
{OP <decl-de-operação>}
DEFINE <lista-de-operações>
UTILIZE <lista-de-operações>
EXTERNAL (<nome-de-arquivo>);
ENDSERVER <nome-su>;
```

O termo <nome-MOD> pode ser um INTERFACE ou outro SERVER. O termo <modo-de-invocação> índica de que modo pode ser chamada uma operação, se é por comando CALL ou SEND, ou via interrupção de interface (INTR-similar à opção SEND). Se não é especificado, fica implícito que é ambos (CALL e SEND). Por exceção entende-se erros que ocorrem durante o tempo de execução, estes são pré-definidos e podem ser:

NUMERROR: Quando o resultado de uma operação numérica pré-definida não cai dentro do intervalo implementado, ou for indefinido. Por ex.: Overflow, underflow, operação inválida, divide por zero.

RESERROR: Quando uma restrição de intervalo ou de índice é violada, ou é feita uma tentativa de acesso de uma capability sem operação associada.

EXECERROR: Quando houver tentativa de executar código com operações incompatíveis em um INTERFACE SOFTWARE ou de executar um procedimento tipo LIB que não esteja na biblioteca ou possua parâmetros incompatíveis.

SELECERROR: Quando todas as alternativas de um comando de seleção sem a opção ELSE estão fechadas, ou é invocada uma operação em um processo já extinto.

ESPERROR: Quando o espaço dinâmico de memória de um processo excede o seu limite.

COMERROR: Quando surge problemas nas comunicações entre processos (operações). Ex: Time--out, parity, etc...

HARDERROR: Qualquer outro erro devido ao equipamento como por exemplo "power failure".

Deste modo, com a opção WHEN, as operações podem ser usadas como rotinas de tratamento de exceções. Nesta opção só podem ser usadas operações que estejam dentro do próprio SERVER.

Com a especificação do intervalo irão ocorrer várias instâncias do mesmo server, assim é possível saber qual a instância por meio da variável MYSERVER.

O termo <nome-de-arquivo> é uma indicação do arquivo onde deve ser procurado o respectivo módulo.

Por <construção-de-ambiente> entende-se não só a construção de ambiente conforme descrito no ítem 3.6, mas também a definição de outros módulos. Nesta construção, quando a declaração CONSTRUCT é omitida, é assumido um ambiente concorrente no hospedeiro ou nodo onde foi localizado este módulo SERVER.

Na declaração UTILIZE, quando é omitida a opção IN, a operação é assumida estar dentro do próprio módulo.

A declaração SHARED indica que a seguir vem um conjunto de procedimentos que podem ser chamados por qualquer processo do módulo. Estes procedimentos são auto-contidos e não podem utilizar variáveis globais, somente os parâmetros e as variáveis internas. Com esta opção é possível implementar o compartilhamento de código de procedimento, pois as

variáveis estarão na pilha de cada processo.

3.7. CLIENTs

O CLIENT é um módulo composto de um conjunto de processos que realizam tarefas. Estes processos se comunicam entre si somente por intermédio de processos em SERVERS, por meio dos quais também acessam recursos do ambiente que compõem. Os CLIENTs não possuem dados permanentes, ou compartilhados. Os CLIENTs também não podem definir operações.

A forma geral do CLIENT é a seguinte:

```
CLIENT <nome-CL>["["intervalo"]"]
[<declaração-de-tipos-e constantes>]
UTILIZE {<nome-operação>[(<parâmetros-formais>)]
                    IN <nome-SY> [WHEN <exceção>]}
<declaração-de-processos>
ENDCLIENT;
```

ou no caso de estar em biblioteca:

```
CLIENT <nome-CL>["["intervalo"]"]
UTILIZE {<nome-operação>[(<parâmetros-formais>)]
          IN <nome-SV> [WHEN <exceção>]}
EXTERNAL;
ENDCLIENT;
```

A descrição dos ítens é idêntica à dos SERVERs. Quando existem várias instâncias de um mesmo CLIENT, então é possível saber qual é a instância pela variável MYCLIENT.

3.8. Interfaces

Conforme o modelo, um Interface é um módulo por meio do qual dois ambientes diferentes podem interagir. É composto de um protocolo, implementado pelos ambientes interagentes, uma formatação de dados e um conjunto de operações.

Na interação entre dois ambientes, via interface, pode haver dois tipos de relacionamento: mestre-escravo e de igualdade. Isto leva, na prática, a três tipos de interface, quanto ao relacionamento: interface mestre, interface escravo e interface simples. Na definição, por exemplo, um interface será mestre quando o ambiente que o contém possui uma relação de escravo com o mesmo, e vice-versa.

O conceito de INTERFACE permite a implementação de comunicação convencional, por meio de protocolos conforme a recomendação ISO [13], entre nodos de uma rede com software completamente diferente.

O interfaceamento com ambientes de hardware apresenta características bem peculiares e diferentes do interfaceamento com ambientes de software. Assim, para efeitos de clareza e concisão de linguagens, considerou-se adequado definir dois tipos de interface: INTERFACE HARDWARE e INTERFACE SOFTWARE.

3.8.1. Interface Hardware

Uma das características peculiares deste tipo de interface é que o mesmo encapsula construções gramaticais especiais para associação de variáveis e endereços reais de hardware. A outra característica é a declaração de procedimentos como co-rotinas para definir a maneira com que as variáveis especiais são acessadas. Com isto é definido o tipo de operação sobre o interface. Finalmente, outra caracte-

rística é a condição de exceção expressa por expressão booleana de variáveis do interface.

As co-rotinas podem ser executadas à maneira das operações SELECT ou como procedimento.

A forma geral do INTERFACE HARDWARE é a seguinte:

```
INTERFACE HARDWARE <nome-int>;
COUPLE <nome-SERVER>;
[<declaração de tipos e constantes>]
VAR {<nome>["["intervalo "]"]:<tipo> AT <endereço
                                        físico>;}
ACESS {<operação com parâmetros reais> WHEN  <ex-
                                    pressão booleana>
[OPER <nome>[(<parâmetros-formais>);
            <corpo-de-procedimento>]
ENDHARD;
```

Onde <endereço físico> também pode considerar arquiteturas que tenham dois espaços de endereçamento: um para memória, e outro para portas de entrada e saída ou de interfaces físicos. Desse modo <endereço físico> pode ser:

```
para memória:  <endereço octal>
              /<endereço hexadecimal>
```

ou para partes:

```
PORT  <endereço octal>
PORT /<endereço hexadecimal>
```

O comando ACESS implementa a interrupção do interface de hardware com a flexibilidade adicional dada pela expressão booleana. Na chamada de operação do SERVER podem ser colocadas quaisquer das variáveis internas como parâmetro real.

Os procedimentos declarados pelo interface podem ser referenciados pelo SERVER concatenando o nome do interface com o nome do procedimento, da mesma forma como a referência de um campo de registro em PASCAL. Assim pode ser chamado como qualquer outro procedimento dentro de um processo.

Cada interface hardware é acoplado a um SERVER, e este SERVER deve estar fortemente conectado com o interface.

3.8.2. Interface Software

A primeira diferença com relação ao INTERFACE HARDWARE é que este não interage apenas com um server do ambiente, mas sim com todo o ambiente que o contém. O INTERFACE SOFTWARE é composto de duas partes: uma que pode interagir com todo o ambiente e outra que interage apenas com um SERVER específico (para fins de monitoração, controle e depuração de programas).

Um programa pode ser carregado para a memória, ou carregar outros códigos de programa, por meio do INTERFACE SOFTWARE. Quando o interface é aquele pelo qual o ambiente referido é carregado, então o mesmo é denominado MASTER, e não declara nenhuma variável e nem possui a segunda parte, que é a de acoplamento de SERVER.

A declaração de variáveis é similar ao do INTERFACE HARDWARE, exceto que não considera o endereçamento de portas, e permite a superposição de variáveis, dependendo do endereço e do tamanho das declarações (redefinição de campos).

De uma forma geral o Interface pode ser:

```
INTERFACE SOFTWARE <nome>["["intervalo"]"]:
MODE {<nome-operação>"["<nodo-de-invocação>"]"}
[UTILIZA {<nome-operação>[(<parâmetros-formais>)]
                IN <nome-MOD>[WHEN <exceção>]:}]
FURNISH {<nome-operação>[<parâmetros-formais>)]}]
<segunda-parte>
ENDSOFT;

Onde <segunda-parte> pode ser:

      MASTER

ou

      COUPLE <nome-server>;
[<declaração de tipos e constantes>]
VAR {<nome>["["intervalo"]"]:<tipo> AT <endereço>;}
ACESS  {<operação com parâmetros reais> WHEN <exp.
                                              bool>;}
[OPER <nome>[(<parâmetros-formais>);
      <corpo-de-procedimento>]
```

Na primeira parte, é gerada uma espécie de capability (V.3.1) com informações adicionais que indicam se a operação é fornecida ou utilizada, e qual o modo de invocação. A responsabilidade da configuração das operações e parâmetros é do código que é carregado logo a seguir da capability do interface, nas variáveis da segunda parte. Se não houver compatibilidade, então ocorre uma exceção de execução (EXECERRO). O código carregado não necessita necessariamente utilizar todas as operações descritas no interface.

São implicitamente definidas as operações SUSPEND, ATIVATE, STOP, EXECUTE e RESTART dentro do INTERFACE SOFTWARE.

No caso de haver declaração de variável para carga de código, é recomendável que toda a área seja primeiro declarada como tipo WORD e depois haja outras declarações. No INTERFACE SOFTWARE a área de memória é ocupada, de forma

contígua, conforme a ordem das declarações, de modo que o código carregado tem condições de saber onde e o que está acessando.

O ítem OPER é o mesmo das co-routinas do interface Hardware e pode ser executado pelo mesmo processador em que executa o SERVER ou pelo processador do interface.

Quando o interface é MASTER, ele é associado com o interface servo do outro ambiente, conforme a conexão na topologia definida em ambos os ambientes. No caso de haver mais de um interface possível de conexão em um mesmo ponto da topologia, então a ordem de codificação é assumida.

Conforme a configuração acima, é possível um SERVER ler um código gerado pelo compilador e distribuir as suas partes conforme a tabela inicial de descrição de topologia, carregando a memória dos vários interfaces software distribuídos em uma rede. O acoplamento entre dois ambientes é de responsabilidade do ambiente mestre que inicializa o interface.

Também é possível carregar código gerado por compiladores de outras linguagens e por em execução, pois o uso das operações do interface não é obrigatório.

4. EXEMPLOS E APLICAÇÕES

A seguir é mostrada a utilização da linguagem na solução de alguns problemas típicos, para dar uma idéia da sua potencialidade. Deve ser observado que os problemas abordados estão longe de explorar todas as potencialidades da linguagem.

Não serão abordados exemplos de aplicação em redes e de tempo real devido a extensão da descrição do problema, necessária para o entendimento do exemplo.

4.1. Construindo mecanismos de sincronização e sequencialização

Antes de entrar nos exemplos é importante ressaltar que a linguagem permite a obtenção do número de invocações pendentes em uma operação por meio da instrução INVOC (<nome-operação>).

A. Semáforos:

```
SELECT P WHEN SEM > 0 DO SEM:=SEM-1
    ALSO V DO SEM:=SEM+1
ENDSEL;
```

Assumindo que P e V sejam invocados pelo comando CALL, a operação acima implementa as operações P e V sobre o semáforo SEM.

Se quizéssemos dar prioridade para as operações P então teríamos:

```
SELECT  P WHEN SEM > 0 DO SEM:=SEM-1
ALSO  V  WHEN(SEM=0 OR INVOC(P)=0) DO SEM:=SEM+1
ENDSEL;
```

A operação V só será executada se, havendo uma invocação pendente para ela, nenhuma operação P pode ser executada (sem=0) ou não tiver nenhuma invocação pendente em P.

B. Monitores:

Como exemplo é apresentado o monitor PAGEBUFFER do sistema SOLO de Brinch Hansen [12]:

```
SERVER Monitor;
OP Read (VAR text[1..512]: CHAR; VAR eof: BOOL):
   Write (VAR text [1..512]:CHAR; eof:BOOL):
DEFINE Read {CALL}, write {CALL}
PROCESS Pagebuffer;
VAR buffer [1..512]:CHAR;
    full,last: BOOL;
WHILE TRUE DO
SELECT Read(text,eof) WHEN full
       DO text:=buffer;
          full:=FALSE
          eof:=last;
ALSO Write (text,eof) WHEN NOT full
          DO buffer:=text;
             full:=TRUE
             last:=eof;
       ENDSEL;
ENDWHILE;
ENDPROS;
ENDSERVER;
```

O monitor pode ser usado como "buffer", interfaceando um Cliente com um "driver" de entrada e saída, por exemplo.

C. Outros

Quando é necessário atender requisição conforme determinada ordem então:

```
SELECT Request (amount) BY  amount
       DO SKIP
   ENDSEL.
```

As requisições pendentes são ordenadas de acordo com o valor de seus parâmetros reais; quando um Request é selecionado, a invocação cujo valor amount for mínimo será executada. Este comando implementa um "shortest-job-next Scheduler", se "amount" representa o tempo máximo de execução ou um "scheduler" por ordem de chegada se amount é um valor de relógio. [5].

4.2. "Rendezvous" Cliente/Servidor

Suponha que existem vários clientes requerendo serviços de um ou vários processos que possuam recursos para fornecer os serviços requeridos [5]. A maneira pela qual ocorre o encontro ("rendezvous") entre estes dois grupos de processos pode ser programada. Quando um cliente necessita um serviço, chama GetService, passando o parâmetro apropriado e recebendo o número de "rendezvous" no retorno. Algum tempo depois o cliente chama Waitdone para esperar a completação do serviço requerido. O número de "rendezvous" é passado como parâmetro.

Solução:

```
SERVER Servo [1..N];
UTILIZE Gettask (task-parâmetros decl.; VAR rendezvous id: INT);
        Taskdone (task-parametros decl.; VAR rendezvous id: INT);
PROCESS Serviço;
      WHILE TRUE DO
         CALL Gettask(task-parameters,rendezvous id);
         faz a tarefa
         SEND TaskDone (result-parameters, rendezvous id);
      ENDWHILE;
    ENDPROS  Serviço;
    ENDSERVER;
    SERVER Rendezvous;
    OP Getservice (Client-params; VAR rid1:INT),
       GetTask (VAR Server-params; VAR rid2:INT),
       TaskDone (results1;rid3:INT);
       WaitDone (VAR results2; rid4:INT);
    DEFINE GetService {CALL},Wait Done{CALL},Gettask {CALL}
           TaskDone;
    PROCESS StartService;
    VAR rid:INT;
     rid:=0
     WHILE TRUE DO
     SELECT GetService (Client-params; rid1) DO
           SELECT Gettask (Server-params; rid2) DO
                  server-params:=client-params;
                  rid1:=rid;
                  rid2:=rid;
                  rid3:=rid+1;
           ENDSEL;
          ENDSEL;
          ENDWHILE;
          ENDPROS Startservice;
```

```
PROCESS Completion;
   WHILE TRUE DO
   SELECT TaskDone (results1; rid3;INT) DO
      SELECT WaitDone (results2; rid4:INT) WHEN
                rid3=rid4  DO  results1:=results2;
       ENDSEL;
      ENDSEL;
   ENDWHILE;
 ENDPROS Completion;
ENDSERVER Rendezvous;
CLIENT tarefas [1..M];
UTILIZE GetServe (Var Chient-params; rendezvous id: INT):
        WaitDone (Var results; rendezvousid: INT)
PROCESS usuário;
        - Procedimentos
          Get Service (client-params, rendezvousid);
        - Outros procedimentos
          WaitDone (results, rendezvouid);
        - Procedimento
ENDPROS Usuário;
ENDCLIENT;
```

No processo StartService irá ocorrer "rendezvous" tão logo haja invocação para GetService e Gettask, durante este encontro será dado um número de "rendezvous" para ambos os processos. O processo Completion sincroniza a completação do serviço e o retorno dos resultados, de acordo com o número de "rendezvous". Conforme programado, nesta solução só é possível processar uma completação de cada vez, podendo assim retardar outros clientes que estejam esperando serviços já realizados. Isto pode ser contornado fazendo com que o processo completion armazene os números de "rendezvouz" dos serviços já realizados, de modo que possa aceitar WaitDone de qualquer tarefa completada.

4.3. "Readers" e "Writers"

Dois grupos de processos, leitores e escritores, compartilham um banco de dados. Para proteger a integridade do banco de dados, no máximo um escritor pode acessá-lo de cada vez, e nenhum leitor pode examiná-lo enquanto um escritor o está alterando; leitores, contudo, podem acessar o banco de dados concorrente. Nenhum leitor ou escritor deve ser postergado indefinidamente. A solução é descrita a seguir [5]:

```
SERVER ReadersWrites;
VAR database [1..N]: item;
OP read (VAR v:ítem; i:INT)[1..M] {CALL}
   write (v:item;i:INT)[1..M]
   startread {CALL}, endread {SEND}, startwrite {CALL}, endwrite {SEND};
DEFINE read [1..M] {CALL}, write [1..M];

PROCESS RW [1..M];
   WHILE TRUE DO
        SELECT read(v,i) DO
                    CALL startread;
                    v:=database [i];
                    SEND endread;
        ALSO write (v,i) DO
                    CALL startwrite;
                    database [i]:=v;
                    SEND endwrite;
        ENDSEL;
   ENDWHILE;
ENDPROS RW;
PROCESS allocator;
VAR state:INT; writelast: BOOL;
    state:=Ø; writelast:=FALSE;
    WHILE TRUE DO
        SELECT startread WHEN (state > 0) AND (write last OR
               INVOC(startwrite)=0) DO
               state:=state+1, writelast:=FALSE;
```

```
        ALSO endread DO state:=state-1;
        ALSO startwrite WHEN (state=0) AND (NOT writelast
             OR INVOC (startread)=0) DO
             state:=-1; writelast:=TRUE,
        ALSO endwrite DO state:=0;
      ENDSEL;
    ENDWHILE;
    ENDPROS allocator;
ENDSERVER Readers Writers;
```

É assumido que existe uma instância J do processo RW para cada usuário do banco de dados. Para acessar o banco de dados, o usuário chama read [J] ou write [J]. O processo allocater força determinados restrições de acesso. A variável "state" indica como o banco de dados, está sendo acessado (state=0 - nenhum leitor ou escritor, state=-1 - um escritor, state > 0 - state leitores); A variável"writelast" indica se um escritor foi o último a acessar o banco de dados. As condições de sincronização das operações do "allocator" asseguram a exclusão requerida, permitem a leitura concorrente e previnem a inanição (postergação indefinida). Um importante atributo desta solução é que encapsula o banco de dados. Nesta solução a granularidade é todo o banco de dados, porém é simples obter a granularidade de um ítem por exemplo. Para isto basta passar o número do ítem como parâmetro nas operações do "allocater" e ao invés de ter uma variável "state", ter um arranjo "state" com um elemento correspondente a cada ítem do banco de dados.

4.4. Transações

No problema anterior foi considerado o caso de leitura ou escrita de um único registro do banco de dados. Contudo, em aplicações de bancos de dados, o processamento de uma transação pode requerer a leitura e escrita de vários registros. Para manter a integridade, é necessário sincroni

zar toda a transação. Primeiro são feitas requisições de leitura ou escrita, depois é feito o acesso ao banco de dados. O pedido de requisições pode ser substituído por uma política de bloqueio ("locking policy") conforme o tipo de acesso desejado.

Solução [5]:

```
SERVER transactions;
  VAR database [1..N]:ítem;
  OP reqread [1..M] {CALL}
     read (VAR v:ítem; i:INT) [1:M]  {CALL}
     relread [1..M], endread [1..M], startread [1..M],
     startwrite [1..M], endwrite [1..M], reqwrite [1..M] {CALL}
     relwrite [1..M],
     write (v:ítem; i:INT) [1..M];
  DEFINE reqread [1..M] {CALL}, read [1..M]  {CALL},
         relread [1..M], reqwrite [1..M] {CALL},
         write [1..M], relwrite [1..M];
  PROCESS R [1..M];
  VAR reading: BOOL;
  WHILE TRUE DO
    SELECT reqread DO
           CALL startread; reading:=TRUE;
    ENDSEL;
    WHILE reading DO
         SELECT read (V,i) DO v:=database [i]
         ALSO  relread DO SEND endread;
                          reading:=FALSE;
         ENDSEL;
    ENDWHILE;

  ENDPROS R;
  PROCESS W[1..M];
  VAR writing: BOOL;
  WHILE TRUE DO
         SELECT  rewrite  DO CALL startwrite;
                          writing:=TRUE;
```

```
            ENDSEL;
            WHILE  writing DO
                   SELECT write (v,i) DO database [i]:=v
                   ALSO relwrite DO SEND endwrite;
                                writing:=FALSE
                   ENDSEL;
            ENDWHILE;
      ENDWHILE;
   ENDPROS W;
   PROCESS allocator;
          (corpo como na seção 4.3)
   ENDPROS allocator;
ENDSERVER Transactions;
```

Esta solução também encapsula o banco de dados, também ilustra a utilidade de aceitar entrada de operações em diferentes partes do corpo do processo. Neste caso assegura que o usuário da transação primeiro faça a requisição, e então acesse o banco de dados, e depois o libere. Também é ilustrado como a construção de um processo pode ser usado para especificar uma família de co-rotinas, uma para cada usuário da transação.

4.5. Escalonamento para otimização de acesso a disco

Seja o problema de utilizar eficientemente uma unidade de disco com braço móvel. Para resolver este problema é necessário reduzir o excesso de movimentos do cabeçote, isto significa que é necessário ordenar a execução de operações de entrada e saída em disco, quando houver mais de uma pendente. Supondo que um dos parâmetros seja c, o índice do cilindro a ser acessado, a solução a seguir seleciona a invocação de operação cujo c seja o mais próximo da posição corrente do cabeçote no disco.

Solução [5]:

```
SERVER disk;
VAR buffers, etc...
OP do-IO (C:INT; ...);
DEFINE do-IO;

PROCESS Driver;
VAR position: INT;
position:=1;

WHILE TRUE DO
   SELECT do-IO(C,...) BY ABS (C-position)
          Position:=C;
          começa I/0 em disco;
          aguarda interrupção do interface;

   ENDSEL;
ENDWHILE;
ENDPROS Driver:
ENDSERVER disk;
```

Embora a solução acima seja utilizada, ela pode levar à postergação de um pedido de entrada e saída para uma posição longe da que está sendo freqüentemente usada.

4.6. "Driver" de console

A seguir é ilustrado o uso de interface de hardware para controle da console. As operações podem ser executadas concorrentemente porque são implementadas por diferentes processos. O exemplo foi tomado de Andrews [8], onde era usado um "Real Resource". A grande vantagem do Interface é a clareza da programação.

```
INTERFACE HARDWARE Intcons;
COUPLE Console;
VAR keyboard: BOOL AT   60;% Keyboard interrupt
                                register
    Printer: BOOL  AT   64;% Printer interrupt
                                register
```

```
        TKS[0..7]: BOOL AT  177560; % reg.status keyboard
        TPS[0..7]: BOOL AT  177564; % reg.status printer
        TKB: CHAR AT  177568; % Keyboard data register
        TPB: CHAR AT  177566; % Printer data register

    ACESS K-Int (TKB) WHEN Keyboard;
    ACESS P-Int       WHEN Printer;
    OPER Startk; TSK[6]:=true, ENDOP;
    OPER Starup; TPS[6]:=true; ENDOP;
    OPER Ack-k; TKS[6]:= false; ENDOP;
    OPEF Ackp; TPS[6]: false; ENDOP;
    OPER Put (C:CHAR);TPB:=C; ENDOP;

ENDHARD;

    SERVER Console:
    CONST N = 80; % tamanho do buffer
    OP getC(RES ch: CHAR) {CALL};
    OP putC(ch: CHAR),
    OP K-Int (TKB:CHAR) {INTR},
    OP P-Int {INTR};
    DEFINE getc {CALL}, putc;
    PROCESS Consin;
    VAR inbuf [0..N-1]: CHAR % Keyboard input buffer
        Busy: BOOL;
        Size, inf, ins: INT;
     Size,inf,ins,busy:=0,0,0,false;
    WHILE size < N AND NOT busy DO
     Intcons.startk; busy:=true;
    ELSE SELECT getc(ch) WHEN size > 0 DO
                ch:=inbuf[inf];
                size,inf:=size-1,(inf+1)MOD N;
         ALSO K-Int(TKB) DO   % interrupt
                Intcons.Ack-k; busy:=false;
                Inbuf[ins]:=TKB;
                size,ins:=SIZE+1, (ins+1) MOD N
         ENDSEL;

    ENDWHILE;
ENDPROS Consoin;
```

```
PROCESS Consout;
VAR outbuf[0..N-1]:CHAR
    Busy: Bool;
    Size, outf, outs: INT;
    Size, outf, outs, Busy:=0,0,0, false;
WHILE size > 0  AND NOT Busy DO
        Intcons.put(outbuf[outf]);
        Intcons.startup; busy:=true;
        Size,outf:=Size-1, (outf+1) MOD N;
ELSE SELECT putc(ch) WHEN Size < N DO
                outbuf [outs]:=ch;
                size,outs:=size+1, (outs+1) MOD: N;
     ALSO p-int DO
                Intcons.Ackp; Busy:=false;
     ENDSEL;
  ENDWHILE;
ENDPROS consout;
ENDSERVER Console;
```

4.7. Carga de código executável

Finalmente é apresentado um exemplo do uso do INTERFACE SOFTWARE, para a carga de programas, e de uso de primitivas do sistema operacional sob forma de operações, por parte do programa carregado. Desta forma o interface também engloba a função do PREFIX do Pascal Concorrente [12].

No ambiente do sistema operacional o interface seria definido da seguinte forma:

```
INTERFACE SOFTWARE Programa;
UTILIZE Read (VAR C:CHAR) {CALL};
       Write (C:CHAR) {CALL};
       etc.
VAR MEM[0..10000]:PAL AT  10200;
COUPLE Loader;
OPER Carga(C:CHAR;i:INT);MEM[i]:=C;ENDOP;
OPER Executa; EXEC (ADDRESS(MEM));ENDOP;
OPER Para; STOP(MEM); ENDOP;
ENDINT;
```

No ambiente do programa carregado teríamos:

```
INTERFACE SOFTWARE Prefix;
MASTER;
DEFINE Read (VAR C:CHAR) {CALL},
      Write (C:CHAR) {CALL};
      etc...
ENDSOFT;
```

Como pode ser visto no exemplo, é bastante simples e clara a definição da interação entre programa usuário e o sistema operacional.

5. ALGUMAS CONSIDERAÇÕES COM RESPEITO À IMPLEMENTAÇÃO

De um modo geral, no projeto do sistema, houve a preocupação de procurar simplificar os problemas de implementação. Assim o tratamento de entrada e saída e de interface é tratado em alto nível, aliviando grandemente o projeto do kernel. Deste modo ao Kernel cabe preocupar-se com a manipulação de interrupções e excessões, do "schedulling" de processos e das operações, da execução da comunicação entre processos, comunicação entre ambientes diferentes, manutenção de tabela de endereços de rede disponíveis, e tabela de estado dos processos de um mesmo ambiente.

Quanto à gramática da linguagem, procurou-se facilitar a construção do compilador. Assim existe um símbolo diferente para a indicação de fim de cada tipo de comando ou módulo. Isto facilita bastante a recuperação de erros no compilador. Também procurou-se facilitar o reconhecimento de estruturas sintáticas (como na declaração de tipo, por exemplo), de modo que não fosse necessário obter mais de um símbolo para saber que tipo de estrutura se trata, durante o reconhecimento (gramática tipo LL [1]).

A linguagem CONSTRUCTOR é um pouco mais complexa que a EDISON [9], para ambientes de processamento paralelo. Para aquela linguagem foi possível construir um compilador, bem menor que o compilador do Pascal Concorrente [12], com 4 passes, ocupando pouca memória. Para o compilador do CONSTRUCTOR também são previstos 4 passes: análise léxica e geração de tabelas, análise sintática, análise semântica, geração e otimização de código.

Os comandos guardados de Dijkstra [7], por definição, são selecionados em uma ordem aleatória quando dentro de um outro comando. No caso da implementação dos comandos da linguagem CONSTRUCTOR, esta ordem será a de escrita.

Ainda é previsto o projeto de um sistema operacional básico que ofereça as funções mínimas necessárias para carga de programas distribuídos. Estas função podem incluir: carga do programa distribuído na rede, manutenção de tabelas de informação sobre o estado de elementos da rede, ou de hospedeiros, operações de entrada e saída, sistema para manuseio de bibliotecas, indicação e/ou recuperação de erros não detectados pelos programas distribuidos carregados. Este sistema básico deve ter condições de conviver com outros sistemas operacionais distribuidos construídos para as várias aplicações específicas possíveis de serem desenvolvidas na rede. Deste modo o desenvolvimento das aplicações é facilitado por uma série de funções de Sistemas Distribuídos já embutidos no Sistema CONSTRUCTOR.

Na implementação do sistema CONSTRUCTOR também haverá uma preocupação com portabilidade. Visto que em uma rede é comum ter hospedeiros de arquiteturas diferentes, é necessário minimizar as dificuldades de implementação, neste sentido, do CONSTRUCTOR em uma máquina. Assim, existem duas possibilidades: a definição de uma máquina virtual (máquina C) para a qual bastaria implementar um interpretador e um kerner no código do hospedeiro, ou o projeto de um compilador "tipo código parcial" onde bastaria trocar a tabela do gerador de código para o código do hospedeiro e programar o kernel no código do hospedeiro. As duas possibilidades podem ser aplicadas concomitantemente ou alternativamente. Como alternativa, é preferido o projeto da máquina virtual, principalmente pelo exito observado com a máquina Pascal Concorrente (MPC).

6. CONCLUSÃO

O sistema CONSTRUCTOR, em projeto atualmente, fornece ao usuário de suporte dois tipos de visão de ambiente: o ponto de vista físico e o ponto de vista lógico. Deste modo o programador de um sistema distribuído deve ter condições de:

- Especificar a distribuição das unidades do programa.
- Estabelecer processos, programando partes que podem ser executadas em paralelo.
- Desenvolver os mecanismos de comunicação entre componentes remotos do programa.
- Providenciar estratégias para o manuseio de excessões.

Estas características introduzem um novo estilo de programação, que é diferente das áreas existentes tais como escrever programas seqüenciais e concorrentes [3].

A organização da linguagem CONSTRUCTOR possibilita uma descrição clara e concisa de qualquer sistema distribuído, e permite inclusive a detecção de vários erros lógicos já durante a compilação. A elaboração do modelo de ambientes antes da definição da linguagem viabilizou o projeto de estruturas tais, que se tornou possível disciplinar a liberdade de uso das potencialidades da linguagem sem tirar a sua versatilidade, como ocorre com o Pascal Concorrente [12], por exemplo. Estas estruturas também introduziram uma nova filosofia de programação, bastante útil para resolver problemas complexos como os de programação distribuída.

7. AGRADECIMENTOS

Ao Prof. Simão Sirineo Toscani pela revisão dos textos originais e pelas sugestões úteis.

8. BIBLIOGRAFIA

[1] ROCHOL, J., NAVAUX, Ph., KEGLEVICH DE BUZIN, P., BRANCO. "Delineamento de um Projeto de Rede Local em Barra na UFRGS". XV Congresso Nacional de Informática-SUCESU Rio de Janeiro-RJ, outubro 1982 (442-446).

[2] Advanced Course on Distributed Systems Architecture and Implementation, München, Mar.4-13, 1980. Edited. Besling, Springer-Verlag, 1981.

[3] TOBIASCH, R. & RAFFLER, H. "Configurating Software a Method for bridging over the gap between concurrent processing and distributing processing". Infocom 82, Nevada, Las Vegas-USA, March-April /1982. (44-48)

[4] WIRTH, Niklaus. "Toward a discipline of Real-time programming". Communications of the ACM, Vol. 20, N.8, August 1977. (577-583).

[5] ANDREWS, Gregory R. "Synchronizing Resources". ACM Trans. on Programming Languages and Systems, Vol. 3, N.4, October 1981. (405-430).

[6] HOARE, C.A.R. "Communicating Sequential Processes". Communications of the ACM, Vol. 21, N.8, August 1978. (666-677)

[7] DIJKSTRA, E.W. "Guarded Commands, Nondeterminacy and formal derivation of Programs". Communications of the ACM, Vol. 18, N.8, August 1975 . (453-457).

[8] ANDREWS, Gregory R. "The distributed Programming Language SR-Mechanisms, Design and Implementation". Software-Practice and Experience, Vol. 12, No.8, August, 1982. (719-753).

[9] HANSEN, P.B. "The Design of Edison" Software-Practice and Experience, Vol.11, No.4, April, 1981 (363-396).

[10] HANSEN, P.B. "EDISON - A Multiprocessor Language". Software - Practice and Experience, Vol. 11, No.4, April, 1981. (325-361).

[11] WIRTH, N. "The Programming Language Pascal". Acta Informatica, Vol. 1, Fasc. 1, 1971 (35-63).

[12] HANSEN, P.B. "The Architecture of Concurrent Programs". Englewood Cliffs, N.J., Prentice-Hall, 1977.

[13] Data Processing - Open Systems Interconnection - Basic Reference Model. Computer Networks 5(1981).81-118.