DESENVOLVIMENTO DE PROGRAMAS DISTRIBUIDOS ATRAVES DE CONFIGURAÇAO DINAMICA DE PROCESSOS

George Roger Ribeiro Justo
Paulo Roberto Freire Cunha

Departamento de Informática
Universidade Federal de Pernambuco
50.739, Recife, PE.

SUMARIO

Este artigo se concentra no estudo e na definição de uma linguagem de programação distribuída baseada em passagem de mensagens e configuração de processos. São estabelecidos alguns conceitos fundamentais e analisadas as principais propriedades de ambientes distribuídos. A linguagem proposta apresentá como principais características a modularização, a criação de módulos reusáveis e a flexibilidade de reconfiguração de sistemas. Destacamos o poder de expressão da linguagem para permitir a construção de configurações mais sofisticadas.

1. INTRODUÇAO

O modelo de arquitetura de máquina de computação idealizado por von Neumann com um controle de execução sequencial e centralizado não tem conseguido atender às novas aplicações exigidas pelos usuários. Já há algum tempo, modelos de arquiteturas alternativos vem sendo projetados buscando aumentar o desempenho no processamento destas aplicações. Podemos destacar entre elas, as máquinas paralelas. Nestas máquinas o processamento é descentralizado através de vários processadores que trabalham cooperativamente [19].

Os sistemas distribuídos, do ponto de vista físico (distribuição física), são uma arquitetura paralela caracterizada pela interligação, através de um meio físico, de máquinas autônomas, ou seja, com capacidade de processamento e de armazenamento próprios. Sob o aspecto de software (distribuição lógica), a propriedade mais marcante dos sistemas distribuídos é a descentralização do controle entre vários componentes de software que executam independentemente e cooperativamente formando o estado global do sistema [15]. Desta maneira, não é possível conhecer o estado global do sistema apenas conhecendo o estado de alguns de seus componentes.

Além do aumento de desempenho os sistemas distribuídos apresentam algumas potencialidades importantes, destacando-se a flexibilidade e a robustez. A flexibilidade é a capacidade do sistema em se adaptar a mudanças do ambiente sem perder a sua funcionalidade, o que implica em não ter que alterar a estrutura básica do sistema. Os sistema distribuídos permitem que componentes possam ser adicionados ou retirados de forma a atender a novos serviços ou novos requisitos de desempenho. A robustez indica a capacidade de funcionamento do sistema após uma falha. Intuitivamente, a maneira mais natural de um sistema se proteger contra falhas é criar e manter um certo grau de redundância. Nos sistemas distribuídos existe um redundância natural, implícita, já que temos vários componentes independentes. Isto torna mais fácil se aplicar mecanismos de recuperação em casos de falhas e permite que o sistema ofereça maior disponibilidade por poder apresentar duplicação de hardware e software.

Embora arquiteturas distribuídas venham sendo desenvolvidas, ainda existem muitos problemas na sua utilização, principalmente relativos à programação de tais ambientes. Neste trabalho, discutiremos alguns pontos referentes à programação de sistemas distribuídos e proporemos um modelo de linguagem. Na seção dois, avaliaremos os principais mecanismos de comunicação e sincronização utilizados em ambientes distribuídos, pois eles são os aspectos fundamentais que caracterizam as linguagens projetadas para estes ambientes. Na seção três, mostraremos algumas linguagens utilizadas em programação distribuída, tratando os aspectos que mais influenciaram nas decisões em relação ao nosso modelo. O modelo de programação deste trabalho será mostrado resumidamente na seção quatro, onde serão basicamente abordados os aspectos que permitem compará-lo com outros modelos existentes. Finalmente, faremos as considerações finais sobre o nosso modelo e apontaremos alguns trabalhos futuros que pretendemos realizada dentro de um projeto para desenvolvimento de ferramentas para sistemas distribuídos.

2. ESTRUTURA DE COMUNICAÇÃO

Analisando os sistemas distribuídos ressaltamos dois aspectos que influenciam fortemente na programação destes ambientes. O primeiro aspecto diz respeito à distribuição física onde se tem um conjunto de máquinas autônomas interligadas. É importante observar que para se programar uma destas máquinas, localmente, não é preciso adicionar

nenhum mecanismo de programação aos existentes. O segundo aspecto é o fato de se ter máquinas interligadas através de um meio físico cooperando na execução de uma tarefa. A fim de se incorporar esta propriedade ao ambiente de programação, temos que adicionar aos mecanismos de programação local recursos que permitam às tarefas cooperantes trocarem informações e definirem pontos de sincronização possibilitando a realização de uma tarefa conjuntamente. Estes dois aspectos devem, então, ser analisados de modo que o ambiente de programação reflita de forma natural o comportamento de um sistema distribuído.

A independência e a dinâmica do ambiente são representadas na programação através de unidades geralmente chamadas de processos ou tarefas. A independência dos processos existe no sentido de que eles contêm toda informação necessária para a sua execução sendo, portanto, autônomos. A comunicação e a sincronização dos processos são geralmente realizadas através de mecanismos baseados em duas filosofias: compartilhamento de variáveis ou passagem de mensagens [4,6,7].

Os mecanismos para comunicação e sincronização baseados em variáveis compartilhadas, tais como semáforos, regiões críticas e monitores, refletem a idéia de sistemas centralizados, ou seja, a existência de um estado global do sistema. Estes mecanismos foram fortemente influenciados pelo modelo de von Neumann, principalmente a conotação global dada ao espaço de memória e a centralização do controle. Os mecanismos baseados em variáveis compartilhadas não são, portanto, aplicáveis no caso de processos que executam em máquinas autônomas interligadas, pois requerem espaços de endereçamento comum.

A comunicação através da passagem de mensagens permite conservar o estado local de cada processo. As mensagens fluem de um processo origem e atingem um processo destino depois de um tempo arbitrário, caracterizando a existência do meio físico de comunicação. Além disso, pode-se observar que a comunicação através de variáveis compartilhadas é um caso específico de passagem de mensagens, onde os dois processos não podem se comunicar ao mesmo tempo, refletindo a exclusão natural daquele tipo de filosofia. Como a passagem de mensagens se aplica mais naturalmente em ambientes distribuídos, decidimos utilizar mecanismos de comunicação e sincronização baseados nesta filosofia.

A definição de mecanismos que permitam a passagem de mensagens requer que consideremos alguns aspectos, tais como os tipos de

transações permitidas e algumas questões referentes ao grau de segurança oferecido pelos mecanismos [17]. Através dos tipos de transações é possível conhecer qual a forma de sincronização que o mecanismo de comunicação oferece, a saber, uma assíncrona e outra síncrona.

a) Transação do tipo notificada (assíncrona):

Este tipo de transação é caracterizado por apresentar o fluxo unidirecional, onde o processo transmissor não exige a indicação de recebimento da mensagem, admitindo que a mensagem sempre é enviada com sucesso. Neste sentido, a sua execução não fica suspensa evitando o problema de bloqueio infinito.

b) Transação do tipo pedido-resposta (síncrona)

Este tipo de transação consiste na transmissão de duas mensagens num fluxo bidirecional. A primeira mensagem é enviada do transmissor para o receptor, enquanto que a segunda traz a resposta do receptor. Uma primitiva de envio deste tipo é síncrona e suspende o processo transmissor até que a resposta seja recebida. O término da comunicação é considerado somente depois do recebimento da resposta pelo transmissor.

Outro aspecto importante de comunicação é o grau de segurança oferecido pelos mecanismos de comunicação. O principal aspecto de segurança de comunicação é a possibilidade de ocorrer bloqueio infinito. Isto ocorre quando uma tarefa espera uma mensagem que nunca é enviada. Os mecanismos de comunicação devem, portanto, incorporar facilidades que evitem este tipo de problema. A solução mais simples é permitir que um processo possa limitar o tempo de espera através da utilização de temporizadores.

Os mecanismos de comunicação são essenciais na avaliação de linguagens para programação de ambientes distribuídos, pois como vimos anteriormente são eles que vão diferenciar a programação distribuída da programação sequencial. Na seção seguinte vamos mostrar as soluções adotadas por algumas linguagens.

3. MODELOS DE LINGUAGENS PARA PROGRAMAÇÃO DE SISTEMAS DISTRIBUIDOS

Recentemente, algumas linguagens de programação de ambientes distribuídos que usam a idéia de estruturação de processos e passagem de mensagens foram propostas. Alguns exemplos representativos destas

linguagens incluem CSP[11], ADA[1,14] e CONIC[9,10,18]. Nesta seção, discutiremos os principais aspectos que caracterizam cada uma destas linguagens.

3.1 CSP ("Communicating Sequencial Processes")

A partir de CSP[11], importantes idéias foram introduzidas na filosofia de programação utilizando processos e mensagens. Um sistema em CSP é um conjunto de processos que se comunicam por meio de comandos de entrada/saída (operações de recebimento e envio). Um comando de saída (envio) em CSP tem a forma

Destino ! Expressão_de_Envio

onde o Destino é o nome de um processo e a Expressão_de_Envio é um valor a ser transmitido. O comando de entrada (recebimento) tem a forma

Fonte ? Variável_de_Recebimento

onde Fonte é um nome de um processo e a Variável_de_Recebimento é uma variável local ao processo que contém o comando de recebimento. Para que dois processos se comuniquem, o valor da Expressão_de_Envio deve ser do mesmo tipo da Variável_de_Recebimento. Este mecanismo de comunicação é síncrono e não existe comunicação assíncrona em CSP.

Um comando de seleção é utilizado para permitir que comandos de recebimento possam ser combinados, podendo ser precedidos por expressões de guarda [8]. Um comando de recebimento com guarda é selecionado para execução somente se a expressão de guarda for verdadeira e o processo destino estiver pronto para enviar os dados. Se várias entradas de um comando de seleção estiverem prontas para receber dados, apenas uma é selecionada arbitrariamente. Os comandos de recebimento podem, também, ser combinados em um comando repetitivo, porém não é permitida a combinação de comandos de envio com comandos de seleção ou repetição.

Utilizando a combinação de comandos de comunicação com comandos repetitivos e seletivos, CSP se mostrou poderosa e versátil na construção de sistemas baseados em processos que interagem através de passagem de mensagens, influenciando as principais linguagens projetadas para programação de sistemas distribuídos posteriores. Como a proposta de CSP não era diretamente voltada para implementação, a grande preocupação das linguagens seguintes foi implementar as idéias propostas por CSP de forma eficiente.

3.2 ADA

ADA[1,14] é o resultado de um esforço coletivo para projetar uma linguagem comum para programação de sistemas de larga escala e /tempo real. Avaliaremos nesta seção, porém, apenas os recursos que a linguagem oferece para permitir programação distribuída.

Os processos são definidos através de unidades de programas chamadas de TASK (tarefa). Uma tarefa contém declarações de ENTRIES (entradas) que são semelhantes às entradas de procedimentos, sendo utilizadas para sincronização e comunicação entre tafefas. A forma básica de comunicação entre tarefas é através da chamada de uma entrada. A chamada de uma entrada é semelhante à chamada de um procedimento, a diferença aconteçe somente no comportamento interno. Assim, nos procedimentos, as ações correspondentes dadas pelo corpo do procedimento são executadas quando o procedimento é chamado. Nas chamadas de entradas, as ações correspondentes são dadas por um comando de ACCEPT e só são executadas quando a tarefa chamada está preparada para aceitar o pedido feito numa entrada. Por outro lado, se uma tarefa está pronta para receber um pedido, porém este não foi ainda realizado, a tarefa fica esperando que ocorra a chamada da entrada. Assim, a tarefa chamada e a que chama para se comunicarem são consideradas como se encontrando em um "rendezvous". A tarefa que chama a entrada tem a sua execução temporariamente suspensa até que a tarefa chamada complete a execução da seqüência de comandos delimitada no ACCEPT e que, assim, os parâmetros de saída sejam passados de volta, permitindo a partir daí que as duas tarefas possam prosseguir independentemente. Este tipo de mecanismo obriga que a tarefa que chama a entrada conheça o nome da tarefa chamada, bem como o nome da entrada. Por outro lado, a tarefa chamada aceita pedidos de quaisquer tarefas, caracterizando a transferência como multi-recebimento. Do ponto de vista do tipo de transação é fácil observar que este tipo de mecanismo é do tipo síncrono. ADA, como CSP, não oferece nenhuma primitiva com transação do tipo assíncrono.

Existe um mecanismo que permite o recebimento de pedidos através de várias entradas, de modo que possa-se ter mais de um ACCEPT sem determinar qual será atendido. O mecanismo é o comando SELECT, semelhante ao de CSP. Se nenhuma entrada for chamada, a tarefa fica suspensa até que uma delas seja chamada. Por outro lado, se mais de uma entrada for chamada uma delas é escolhida arbitrariamemte. O comando

SELECT permite, também, que sejam utilizadas expressões de guarda que devem ser avaliadas sendo consideradas como elegíveis apenas aquelas entradas cujas condições guardiãs sejam verdadeiras. A forma geral do comando SELECT é a seguinte:

```
SELECT
   WHEN Expressão_de_Guarda1 =>
        Comando ACCEPT ...
   WHEN Expressão_de_Guarda2 =>
        Comando ACCEPT ...
        .......
   WHEN Expressão_de_GuardaN =>
       Comando ACCEPT .....
   OR   Comando ACCEPT ...
   ELSE
        Sequência_de_Comandos
   END select;
```

Note que existe uma parte ELSE no SELECT, sua função é evitar que uma tarefa fique bloqueada caso não exista nenhuma entrada pronta para realizar um "rendezvous", ou seja, funciona como um temporizador implícito com tempo de espera zero. Na verdade, na semântica do SELECT, o ELSE apresenta um duplo sentido, além de temporizador ele se apresenta também como alternativa das expressões de guarda, onde a parte ELSE é executada se nenhuma expressão de guarda for verdadeira. Assim, surge uma dificuldade porque, como o ELSE representa um temporizador zero, não podemos utilizá-lo juntamente com outro temporizador, mesmo que o ELSE esteja sendo utilizado como uma alternativa das expressões de guarda. Isto ocorre porque não podemos separar os dois significados dados a ele. Então, achamos que este duplo sentido dado ao ELSE restringe a capacidade do SELECT.

3.3 CONIC

CONIC[9,10,18] foi especialmente projetada para atender à programação distribuída, incorporando características para solucionar vários problemas referentes a este tipo de programação. A linguagem fornece uma das propriedades essenciais dos sistemas distribuídos, a flexibilidade. Esta propriedade permite que o sistema englobe e possa ser modificado para atender a novos requisitos e condições, como por exemplo, mudanças evolutivas permitindo que componentes possam ser

fisicamente realocados. CONIC utiliza a modularidade como elemento principal para fornecer flexibilidade. Então, um sistema pode ser decomposto em pequenas unidades chamadas módulos TASK (módulos tarefa), que são compiladas e testadas independetemente e o sistema é, então, construído através da conexão destes módulos. CONIC apresenta uma separação nítida entre estes dois passos, de forma que existe uma linguagem para programação dos módulos tarefas, a Linguagem de Programação de Módulos[9] e outra para configuração do sistema através da ligação dos módulos, a Linguagem de Configuração[10]. Veremos a seguir as características principais das duas linguagens.

4.2.1 A Linguagem de Programação de Módulos

Esta linguagem permite definir um módulo tarefa como um processo sequencial e auto-contido independente de como será configurado. Esta característica conhecida como independência de configuração obriga que todas as referências sejam feitas a objetos locais e que não possa haver chamada direta a outros módulos. Na verdade, os módulos contêm uma interface bem definida especificando o tipo de informação que o módulo pode trocar com meio externo. A comunicação é feita através de portas. Uma EXITPORT (porta de saída) denota a interface onde o módulo pode enviar informação para o meio externo, enquanto que o módulo recebe informação através de uma ENTRYPORT (porta de entrada). Assim, os módulos utilizam apenas nomes locais para se comunicarem.

A linguagem de programação foi definida como uma extensão do PASCAL ISO, adicionando as construções para permitir a passagem de mensagens. As portas são definidas indicando o tipo de dado que poderá ser transferido. Como PASCAL apresenta uma checagem de tipos forte qualquer informação que for enviada ou recebida de uma porta deverá ser avaliada para se certificar que é compatível com o tipo da porta indicada. Esta característica ajuda a tornar a comunicação ainda mais confiável.

Os comandos de passagem de mensagens permitem os dois tipos de transações, a notificada e a pedido-resposta. Deve-se lembrar que os valores enviados e os recebidos devem ser do mesmo tipo da porta associada. O envio do tipo notificado fornece comunicação unidirecional e assíncrona. O envio de mensagens do tipo pedido-resposta fornece uma comunicação bidirecional e síncrona. O transmissor fica suspenso até que a resposta seja recebida.

O recebimento do tipo pedido-resposta deve ser seguido de um comando REPLY contendo a resposta que será enviada ao transmissor. A execução deste comando não precedida do comando de RECEIVE se torna sem efeito. A tarefa que executa o REPLY não fica suspensa.

Existe, também, um mecanismo que permite que o recebimento de mensagens possa ser combinado numa seleção como em ADA. A parte ELSE do SELECT apresenta o sentido duplo como em ADA e, portanto, os mesmos problemas.

4.2.2 A Linguagem de Configuração

Os sistemas em CONIC consistem de instâncias de módulos interconectadas. A especificação da configuração identifica os módulos que constituirão o sistema, declara as instâncias destes módulos e descreve a interconexão das instâncias pela ligação entre as portas de saída a portas de entrada. As instâncias de módulos são criadas a partir dos módulos tarefas definidos.

A linguagem de configuração é simples e tem basicamente três tipos de mecanismos, um para permitir a definição do contexto, um para criação de instâncias de módulos e outro para ligação das portas.

CONIC permite mudança de configuração, ou seja, um sistema pode ser reconfigurado dinamicamente durante a sua execução, sem ser necessária a parada completa do sistema. Assim, modificações e extensões podem ser feitas em sistemas existentes sem ter-se que reconstruir todo o sistema. As mudanças são realizadas submetendo-se uma especificação de mudança ao sistema que valida a mudança e produz uma nova especificação do sistema incorporando as mudanças.

A linguagem de configuração não funciona realmente como uma linguagem de programação procedural usual, permitindo tomar decisões em tempo de execução quanto à forma de configuração do sistema. Portanto, o sistema deve ter uma estrutura fixa, podendo apenas ser alterada se uma nova especificação for executada.

4. MODELO DE LINGUAGEM PROPOSTO

A nossa prosposta foi influenciada por CONIC, pois esta linguagem fornece importantes propriedades de sistemas distribuídos, ou seja, independência, já que os módulos não compartilham nem nomes como ocorre na chamada de procedimentos, e flexibilidade, através da separação da programação dos módulos e da configuração que permite alterar a estrutura do sistema para atender novos requisitos. A

utilização de módulos que contêm interfaces bem definidas permite o reuso de software através da criação de instâncias. Então os sistemas são construídos a partir da criação de instâncias de módulos existentes conectados de acordo com a configuração.

O principal motivo de criarmos um novo modelo a partir de CONIC foi propor uma solução para a principal deficiência da linguagem, a saber, o baixo poder de expressão da Linguagem de Configuração. Além disto, propusemos alguns alterações nas primitivas a fim de melhorar a estruturação. O problema do baixo poder de expressão existe, na verdade, na maioria dos modelos que apresentam separação entre programação de módulos e linguagem de configuração, pois geralmente não existe preocupação em oferer mecanismos que permitam a descrição de configurações mais complexas de maneira mais eficiente. Por isso, o usuário é obrigado muitas vezes a realizar várias operações manuais, como por exemplo em CONIC onde deve-se criar uma especificação de mudança e submetê-la ao sistema como única forma de reconfiguração.

Veremos nesta seção a forma geral do nosso modelo de linguagem. Não serão abordados todos os detalhes da linguagem, mas apenas as carecterísticas principais que permitem avaliar a sua aplicabilidade na programação distribuída.

4.2 PROGRAMAÇÃO DE MODULOS

A linguagem para a descrição dos módulos será definida como uma extensão do PASCAL, sendo adicionados os mecanismos necessários para a programação distribuída. A interface dos módulos é definida através de portas de comunicação. A declaração de uma porta deve indicar qual a direção da transferência, o tipo de dado válido e o tipo de transação.

```
EX: EXITPORT    x : integer;  (porta de saída assíncrona)
    ENTRYPORT   y : integer;  (porta de entrada assíncrona)
    EXITPORT    b : integer REPLY c : real; (porta de saída síncrona)
    ENTRYPORT   d : integer REPLY c : real; (porta de entrada síncrona)
```

Uma porta de saída só pode ser conectada a uma porta de entrada com mesmo tipo de dado e mesmo tipo de transação. A comunicação é realizada enviando ou recebendo dados para/de uma porta. As primitivas de comunicação estão agrupadas de acordo com o tipo de transação:

a) Transação Notificada

O envio assíncrono não suspende o transmissor e pode ser

multidestino se a porta de saída estiver conectada a várias portas de entrada. A forma deste comando é a seguinte:

```
SEND Expressão_Mensagem TO Porta_de_Saída
```

A Expressão_Mensagem é avaliada e o seu tipo comparado com o tipo da Porta_de_Saída. Se houver incompatibilidade de tipo, então ocorre erro. No recebimento de dados de uma transação notificada o módulo fica suspenso até que chegue uma mensagem na porta de entrada especificada. A forma geral desta primitiva é a seguinte:

```
RECEIVE Variável_Mensagem FROM Porta_de_Entrada;
```

A Variável_Mensagem deve ser compatível com o valor declarado para a Porta_de_Entrada.

b) Transação pedido-resposta

Neste tipo de envio o módulo transmissor, após enviar a mensagem, tem a sua execução suspensa a espera de uma mensagem de' resposta do receptor. Veja a seguir a forma deste comando:

```
SEND   Expressão_Mensagem  TO  Porta_de_Saída
         WAIT  Variável_Mensagem [ => seqüência_de_comandos ]
         [ FAIL [Temporizador] => seqüência_de_comandos ];
```

A cláusula FAIL é utilizada para testar alguma falha no envio da mensagem ou no recebimento da resposta, sendo o tempo de espera limitado pelo temporizador. O recebimento de uma mensagem numa transação pedido-resposta é semelhante ao ACCEPT no "rendezvous" de ADA. O módulo fica bloqueado a espera de uma mensagem. Depois da chegada de uma mensagem é executada uma seqüência de comandos e, então, enviada a resposta ao transmissor.

```
RECEIVE Variável_Mensagem  FROM  Porta_de_Entrada
          => seqüência_de_comandos
REPLY     Expressão_Mensagem;
```

Preferimos utilizar o mecanismo de resposta (REPLY) como uma cláusula do RECEIVE e não um comando como ocorre em CONIC. O nosso objetivo foi apresentar um comando estruturado, onde existe a amarração entre o RECEIVE e o REPLY. Em CONIC, o REPLY corresponde a um comando independente e, assim, pode ser utilizado em qualquer parte do programa, podendo inclusive ser colocado sem um RECEIVE correspondente, tornando-se um comando sem efeito (comando nulo). Além de oferecer uma

as alterações, não sendo possível já definir na especificação que o sistema tenha uma reconfiguração automática.

A nossa solução é baseada no mesmo princípio aplicado na Linguagem de Programação de Módulos onde se utilizou uma linguagem procedural já existente e se adicionou os mecanismos necessários para a comunicação. No caso da linguagem de configuração, a idéia foi utilizar uma linguagem já existente e introduzir os mecanismos que permitissem a configuração. Para manter a padronização, utilizamos o PASCAL como a linguagem base.

As Linguagens de Configuração são, geralmente, formadas apenas pelos comandos de configuração, restringindo o poder de expressão da linguagem, pois estes comandos não permitem a construção de configurações mais complexas. Para ilustrar podemos dar um exemplo simples que mostra que a Linguagem de Configuração de CONIC não apresenta uma forma natural para solucionar certos problemas, mesmo simples. Considere um sistema composto de n máquinas (estações) conectadas onde cada uma das estações controla um conjunto de equipamentos quaisquer. Desejamos construir um sistema de supervisão composto de n módulos alocados em cada uma das máquinas. Cada módulo faz uma varredura nos valores de variáveis dos equipamentos submissos à estação e os envia a um módulo gerente que gera um relatório estatístico da situação dos equipamentos. Por uma questão de segurança, o módulo gerente é colocado aleatoriamente em uma das estações cada vez que o sistema é iniciado. Em CONIC, a configuração deste sistema deveria ser feita de uma das seguintes formas:

i) Gerando-se n configurações diferentes, onde em cada uma o nó i, no intervalo (1..n), seria considerado como o nó gerente, ou

ii) alterando a especificação da configuração do sistema, manualmente, antes de cada iniciação do sistema, modificando o nome da estação onde o módulo gerente deve ser colocado.

É fácil observar que as duas soluções são ineficientes e são resultado do baixo poder de expressão da Linguagem de Configuração. No nosso modelo, a configuração pode ser melhor modelada, como por exemplo:

i) utilizando um programa que calcula, através de uma função, em qual estação será alocado o módulo gerente e automaticamente define a configuração, ou,

ii) através de um programa que interage com o usuário e pede para

indicar em qual estação será colocado o módulo gerente.

Comparando as duas soluções, podemos notar que a idéia do nosso modelo é permitir que a configuração possa ser flexível e expresse mais naturalmente o comportamento do sistema sendo representado.

Além de permitimos a utilização do PASCAL nós, também, adicionamos novos mecanismos à Linguagem de Configuração e alteramos alguns dos existentes em CONIC. A configuração é baseada em quatro etapas, a saber, a definição de contexto, a criação de instâncias de módulos, a ligação de portas e a ativação de uma instância, enquanto que a reconfiguração é realizada através da desativação de instâncias, da desconexão de portas, da destruição de instâncias e da redefinição de contexto.

Na parte de definição de contexto se declaram os módulos que serão utilizados na configuração. Veja a seguir:

```
USE Id_de_Módulo [, Id_de_Módulo ]* ;
```

Só podemos criar instâncias de módulos declarados no contexto do sistema. Uma instância de um módulo é criada definindo o seu nome, indicando o nome do módulo a partir do qual será criada e a estação onde será criada.

```
CREATE Id_de_Instância
       FROM Id_de_Módulo
       AT   Id_de_Estação;
```

A cláusula FAMILY pode ser utilizada no comando CREATE para permitir que uma família de instâncias (array) de um mesmo módulo possa ser criada. Por exemplo:

```
CREATE FAMILY k:[1..n] janela[k]
       FROM   gerencia_janela;
```

A ligação de portas permite que uma porta de saída possa ser conectada a uma porta de entrada e, assim, que os dois módulos possam se comunicar. A cláusula FAMILY permite que um conjunto de portas possa ser conectado definindo comunicação múltipla, bem como para facilitar a conexão de famílias de instâncias. Por exemplo:

```
LINK FAMILY k:[1..n]
     janela[k].escreve TO escreve_console;

LINK FAMILY k:[1..n]
     servidor.porta_servico[k]   TO   cliente[k].resposta;
```

O comando ACTIVATE é utilizado para ativar uma instância , ou seja, iniciar explicitamente a sua execução. A utilização da cláusula

FAMILY neste comando permite que um conjunto de instâncias do mesmo nome possa ser ativado. Vejamos um exemplo do comando ACTIVATE:

ACTIVATE Id_de_Instância [, Id_de_Instância]* ;

ACTIVATE FAMILY k:[1..n] janela[k];

Veremos agora os comandos que permitem modificar uma configuração já definida. Uma instância em execução pode ser cancelada através de um comando contrário ao ACTIVATE. Como no caso do ACTIVATE, pode-se utilizar a cláusula FAMILY para desativar famílias de instâncias:

DEACTIVATE Id_de_Instância [, Id_de_Instância]* ;

DEACTIVATE FAMILY k:[1..n] janela[k];

O comando de desconexão é utilizado na reconfiguração de sistema e o seu efeito é desligar a conexão existente entre uma porta de saída e outra de entrada. Através deste comando (UNLINK) fica encerrada a comunicação entre as instâncias. Seu efeito é exatamente contrário ao do comando LINK.

Outro mecanismo de reconfiguração é o comando de destruição de instâncias. Através deste comando instâncias deixam de fazer parte do sistema. Assim, fica proibida a ativação ou conexão de portas desta instância. Uma instância só poderá ser destruída se todas as suas portas estiverem desconectadas e a sua execução interrompida. O efeito deste comando é inverso ao do CREATE.

Ex: DELETE Id_de_Instâncias [, Id_de_Instâncias]* ;

DELETE FAMILY k:[1..n] janela[k];

Existe, ainda, o comando que permite redefinir o contexto, ou seja, retirar módulos que fazem parte da estrutura do sistema. Depois de retirado do contexto, nenhuma instância do módulo poderá ser criada. Um módulo só poderá ser retirado do contexto após a destruição de todas as suas instâncias. É fácil notar que este comando é contrário ao USE. A forma do comando é a seguinte:

REMOVE Id_de_Instância [, Id_de_Instância]* ;

É interessante observar que os comandos de reconfiguração seguem uma ordem, ou seja, desativação de instâncias, desconexão das portas, destruição de instâncias e finalmente a mudança no contexto. Esta ordem é importante para não se ter que incorporar a semântica do comando anterior à semântica do comando seguinte. Além disso, esta restrição possibilita fazer uma correspondência clara entre os comandos de configuração e de reconfiguração. Porém deve ficar claro que esta ordem

diz respeito a um dado módulo, suas instâncias e as ligações destas instâncias. Esta observação é importante porque em CONIC todos os módulos são introduzidos de uma única vez, em seguida todas as instâncias são criadas e depois são definidas as ligações. Assim, existe uma ordem rígida e não podemos , por exemplo, criar instâncias e depois introduzir módulos no contexto. Esta rigidez é necessária , pois como existem apenas estes comandos na linguagem a mistura deles só levaria a uma má estruturação. No nosso modelo, porém, como podemos criar configurações mais sofisticadas esta restrição de CONIC diminuiria a capacidade da linguagem, pois podemos, por exemplo, desejar que uma instância só seja criada após uma certa condição, enquanto que outras já podem estar em execução.

Definimos, ainda, um comando especial que permite que o programa de configuração possa ficar suspenso a espera do final da execução de um módulo. A forma do comando é a seguinte:

```
WAIT Id_de_Instância [=> Seqüência_de_Comandos]
     [Temporizador => sequência_de_Comandos];
```

onde o Id_de_Instância indica o nome da instância de módulo pela qual o programa de configuração deve esperar, e o Temporizador permite limitar o tempo de espera máximo. Este comando facilita a construção de sistemas cuja configuração muda após a execução de certos módulos. Geralmente, isto só seria possível manualmente, onde o usuário deveria esperar o término da execução do módulo para submeter a nova especificação de configuração.

Entendemos que com a utilização de todas as potencialidades de PASCAL e mais com os novos mecanismos de configuração (ACTIVATE, DEACTIVATE e WAIT) agora podemos descrever configurações bem mais flexíveis e eficientes para ambientes distribuídos.

5. CONCLUSOES

Embora as arquiteturas distribuídas já estejam sendo projetadas, um grande problema ainda existente é a pouca disponibilidade de linguagens que permitam projeto de software que utilizem eficientemente tais ambientes. Assim, quaisquer novas tentativas devem ser estimuladas a fim de que as experiências possam trazer soluções para muitos problemas ainda existentes.

O nosso modelo foi influenciado por CONIC porque esta linguagem oferece bons recursos que permitem criar sistemas com contro.e

distribuído, principalmente permitindo uma separação entre a programação dos módulos e de sua configuração. Quanto aos mecanismos de comunicação foi possível observar que em termos de filosofia das primitivas de comunicação não existe muita distinção na maioria dos modelos. Algumas linguagens omitem algum tipo de primitiva ou permitem utilizar uma facilidade a mais, porém em geral existe um certo consenso.

Um ponto importante da programação distribuída é sem dúvida a separação entre a programação dos módulos e de sua configuração. Nós vimos que esta característica incorpora propriedades importantes dos ambientes distribuídos, já que os módulos realmente são independentes e se comunicam sem compartilhar nomes ou código. Tentamos incorporar à linguagem de configuração mecanismos que permitissem não apenas separar a configuração da programação de módulos, mas programar como será a configuração do sistema. Podemos observar, então, a grande capacidade do nosso modelo na proposição de soluções para problemas de ambientes distribuídos.

Nos pretendemos desenvolver um protótipo do nosso modelo no ambiente VAX/VMS[20,21] a fim de podermos verificar como o modelo se comporta na solução de alguns problemas típicos de ambientes distribuídos, detalhes podem ser vistos em [12]. Embora este ambiente não seja distribuído, como o modelo oferece uma "transparência de localização", onde não importa em que estações os módulos estejam localizados achamos que isto torna possível uma boa avaliação. Além disso, o ambiente VAX/VMS apresenta as principais propriedades para a aplicação da estruturação multi-processos [4], a saber:

a) A criação de processos baratos:

Como a idéia é explorar o uso de muitos processos é importante que o sistema garanta que isto não comprometa o bom funcionamento do sistema. Por exemplo, um sistema que utiliza uma grande área de armazenamento na definição de um processo não estimula o uso desta metodologia.

b) Identificação eficiente de processo:

Os nomes dos processos vão ser utilizados várias vezes na criação, comunicação, etc. Então, é essencial que seja simples o mecanismo de identificação de um processo.

c) Configuração dinâmica:

Pode ser possível alocar recursos dinamicamente, bem como liberá-los. Assim, a criação e a destruição dinâmica de processos é necessária desde que um processo é um recurso.

É importante observar que existe uma semelhança na filosofia de estruturação multi-processos e a utilização de programação de módulos e de configuração. A estruturação multi-processos explora a idéia de estrutura de programas como vários processos concorrentes. Esta metodologia impõe uma estruturação de um programa através de processos e suas interações, principalmente aquelas via passagem de mensagens, onde cada processo corresponde a uma função ou gerencia um recurso do programa. Além disso, é possível modificar a estrutura do programa dinamicamente pela criação e destruição de processo. Neste sentido, pretendemos explorar as propriedades de estruturação multi-processos para diminuir os esforços de implementação do nosso protótipo, já que o ambiente VAX/VMS permite a utilização desta metodologia.

REFERENCIAS

[1] "ADA Reference Manual"; United States Departament of Defense, Jul/1980.

[2] Andrews, G.R.: "Synchronizing Resources"; TR 78-360, Department of Computer Science, Cornell University, Ithaca, Fev/1979.

[3] Andrews, G.R.:"Synchronizing Resources":ACM Transactions on Programming Languages and Syatems, Vol. 3, No. 4, University of Arizona, Out/1981.

[4] Andrews, G.R.; Schnieder, F.B.: "Concepts and Notations for Concurrent Programming"; Computer Surveys, Vol. 15, No. 1, Mar/1983.

[5] Cheriton, D.R.: "Multi-Process Structuring and the THOTH Operating System"; Ph.D thesis, University of Waterloo, 1979.

[6] Cunha, P.R.F; Lucena, C.J.; Maibaum, T.S.E.: "On the Design and Specification of Message Oriented Programs"; International Journal of Computer and Information Science, Vol. 9, No. 3, Jun/1980.

[7] Cunha, P.R.F; Maibaum, T.S.E.: "Resource = Abstract Data Type + Synchronization, a Methodology for Message Oriented Programming"; Fifth International Conference on Software Engineering, IEEE Press, 1981.

[8] Dijkstra, E.W.: "Guarded Commands, Nondeterminancy, and Formal Derivation of Programs"; Communications of ACM, Vol. 18, No. 8, Ago/1975.

[9] Dulay, N.; Kramer, J.; Magee, J.; Sloman, M.; Twidle, K.: "The CONIC Programming Language, version 2.4"; **Research Report DOC 84/19**, Department of Computing, Imperial College, Out/1984.

[10] Dulay, N.; Kramer, J.; Magee, J.; Sloman, M.; Twidle, K.: "The CONIC Configuration Language, Version 1.3"; **Research Report Doc 84/20**, Department of Computing, Imperial College, Nov/1984.

[11] Hoare, C.A.R.: "Communicating Sequencial Processes", **Communications of the ACM**, Vol. 21, No. 8, Ago/1978.

[12] Justo, G.R.R, Cunha, P.R.F.: "Modelo de Programação Distribuída Baseado numa Linguagem de Configuração de Processos"; **Relatório Técnico RT-DI/UFPE 015/87**, Departamento de Informática, UFPE, Dez/87.

[13] Kramer, J.; Magee, J.: "Dynamic Configuration for Distributed Systems"; **IEEE Transactions on Software Engineering**, Vol. 11, No. 4, Abr/1985.

[14] Ledgard, H.: "ADA an Introduction"; **Springer Verlag**, 1980.

[15] Melnikoff, S.S.: "Sistema AP^2: Um Ambiente para Programação Paralela"; **Tese de Doutorado**, Universidade de São Paulo, 1982.

[16] Queiroz, R.J.G.B.; Cunha, P.R.F.: "Uma Análise da Programação como uma Atividade de Expressão do Raciocínio"; **Revista Brasileira de Computação**, Vol 4, No. 3, 1984/1985.

[17] Sloman, M.; Kramer, J.; Magee, J.: "A Flexible Comunication Structure for Distributed Embedded Systems"; **Research Report DOC 83/11**, Department of Computing, Imperial College, Abr/1984.

[18] Sloman, M.; Kramer, J.; Magee, J.: "The CONIC Toolkit for Building Distributed Systems"; **6th IFAC Distributed Computer Control System Workshop**, Mai/1985.

[19] Tanenbaum, A.S.; Renesse, R.: "Distributed Operating Systems", **Computing Surveys**, Vol. 17, No. 4, Dez/1985.

[20] "VAX/VMS System Services Reference Manual, Version 3.0"; **Digital Software**, Vol. 5A, 1982.

[21] "VAX Architecture Handbook", **Digital Press**, 1981.