Suporte à Execução do Ambiente DisCo

Sílvio Soares Bandeira *Paulo R. F. Cunha*

Universidade Federal de Pernambuco
Departamento de Informática
Caixa Postal 7851
50732-970 Recife-PE
E-mail: {ssb,prfc}@di.ufpe.br

Resumo

Sistemas distribuídos oferecem uma arquitetura de implementação atrativa para muitas aplicações, particularmente para aquelas fisicamente distribuídas. Neste artigo, apresentamos a implementação do ambiente de suporte do ambiente DisCo, para construção e gerenciamento dinâmico da configuração de aplicações distribuídas. Relatamos informações detalhadas sobre seu sistema de comunicação e suporte a configurações. Uma discussão sobre avaliação do desempenho do sistema encontra-se no final do artigo.

Abstract

Distributed systems offer an attractive implementation architecture for many applications, particularly those environments where the application is itself physically distributed. In this paper, we present the implementation of DisCo's run-time support environment, for the construction and dynamic management of distributed systems configuration. We have included detailed informations about its communication system and configuration support. A discussion concerning performance evaluation is also provided at the end of the paper.

1 Introdução

A favorável relação custo/desempenho proporcionada pelos sistemas distribuídos é uma dentre as várias vantagens que estes sistemas podem suprir. Outras características dos sistemas distribuídos contribuem definitivamente para o crescente interesse nesta área. Algumas delas são **paralelismo**, **modularidade**, **flexibilidade** e **tolerância a falhas**, aumentando a **disponibilidade** e **confiabilidade** das aplicações.

Dentre os vários segmentos dos sistemas distribuídos, o paradigma de configuração desempenha um papel de destaque. Neste paradigma, a aplicação é vista como uma distribuição topológica dos seus processos, e é estruturada separando-se a definição desta topologia da programação dos seus componentes básicos.

Neste artigo, descrevemos a implementação do ambiente de execução do ambiente *DisCo*. Este ambiente foi concebido no Departamento de Informática da Universidade

Federal de Pernambuco e adota o paradigma de configuração como metodologia para descrição, construção e evolução de sistemas [JC92, JCdP93]. A linguagem CL [dP93, CndP93], também desenvolvida neste departamento, é a linguagem de configuração do ambiente DisCo.

O ambiente DisCo se divide em três partes: ambiente de compilação da linguagem CL, ambiente de suporte gráfico [dS94], e ambiente de execução distribuído [Ban94].

1.1 Ambiente de Compilação

Para a compilação e validação das especificações escritas na linguagem CL, dispomos de um ambiente de compilação [dP93]. Este ambiente de compilação é subdividido em:

1. Decodificador de classes

   Este decodificador faz todas as verificações referentes às classes de portas e monta a interface do módulo da forma como o meio exterior e o ambiente de execução a enxergam. Utilizamos o termo *decodificador* para deixar claro que as classes não passam por um processo de compilação, uma vez que o resultado da decodificação não é um código executável, mas apenas um código intermediário que será usado pelo compilador da linguagem de configuração.

2. Compilador da linguagem de configuração

   Este compilador tem seu analisador semântico baseado na especificação formal da semântica da CL, escrita em Action Semantics [Mos92] e descrita em [dP93].

3. Pré-processador da linguagem dos componentes

   A linguagem dos componentes é a linguagem de programação C acrescida de algumas características, a saber: capacidade de declaração da interface do módulo e comandos que permitam ao módulo enviar dados para uma porta de saída ou ler mensagens de uma porta de entrada. Assim, é possível usar um compilador já disponível no ambiente para a "linguagem-base" e desenvolver, apenas, um pré-processador para tratar comandos acrescidos.

1.2 Interface Gráfica

A interface no ambiente DisCo, é formada por janelas através das quais o usuário pode verificar o estado da aplicação como um todo, de nós da aplicação, das portas dos nós e das interconexões dos diversos nós da aplicação. Desta forma, o usuário dispõe de uma visão da estrutura da aplicação em questão. Alterações provocadas pela reconfiguração da aplicação são refletidas no desenho da estrutura da aplicação. A interface gráfica [dS94] provê, ao ambiente, recursos de navegação (*system browsing*) através da hierarquia dos módulos; permite múltiplas visões da estrutura (*zoom*); e oferece uma animação da estrutura, através da monitoração da aplicação, à medida que as reconfigurações dinâmicas vão sendo efetivadas.

2 Ambiente de Execução

Como podemos ver na figura 1, o ambiente de execução possui três tipos de módulos:

- Gerenciador de Configuração (principal);
- Gerenciadores Auxiliares;
- Servidor de Nomes.

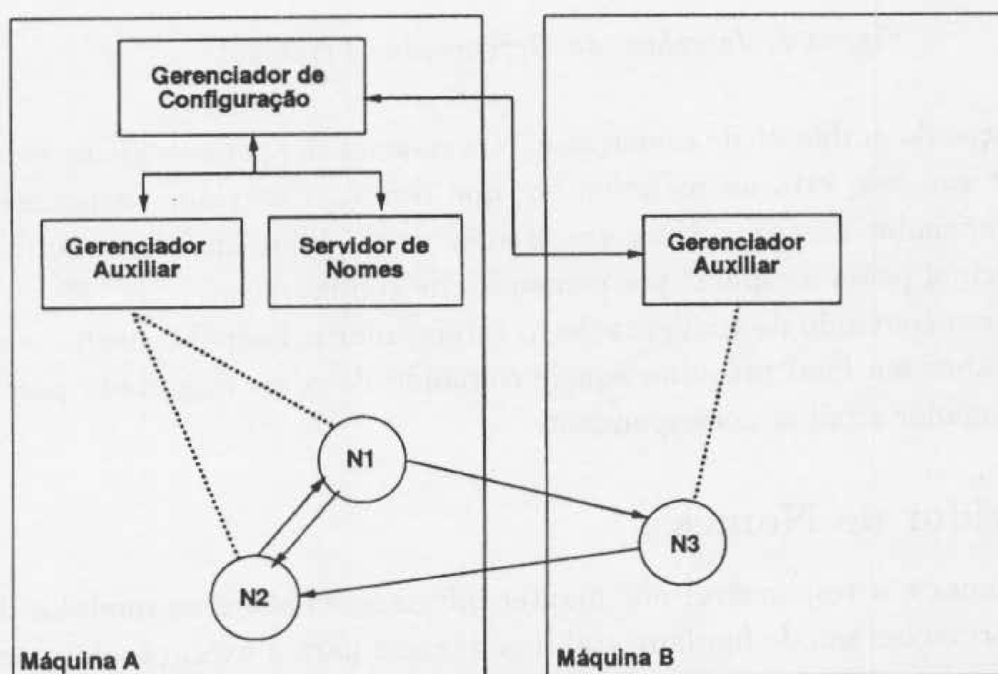

Figura 1: *Estrutura Geral do Ambiente de Execução*

O gerenciador principal recebe os comandos de configuração do ambiente de compilação. Estes comandos devem estar prontos a serem executados e já validados. O ambiente de compilação é o responsável por esta tarefa. Os comandos são recebidos pelo gerenciador principal através de uma porta especial que deve estar conectada ao ambiente de compilação. Uma porta exclusiva para o ambiente de compilação permite que escolhamos um protocolo específico para esta conexão levando em conta as necessidades de segurança e rapidez. Usando uma comunicação orientada a conexão com um protocolo confiável, podemos inclusive detectar uma possível falha no ambiente de compilação pela quebra da conexão.

Como mostra a figura 2, o gerenciador principal possue a porta *commands* usada para este fim. Além desta, temos mais três portas usadas para comunicação com os outros módulos de gerenciamento. As portas *in* e *out* são usadas para comunicação com os gerenciadores auxiliares. Pela porta *out* são iniciadas conexões e pela porta *in* são recebidas conexões destes módulos. A quarta porta, *nserver*, é usada para comunicações com o servidor de nomes. Todas estas portas do gerenciador principal exigem comunicação por conexão.

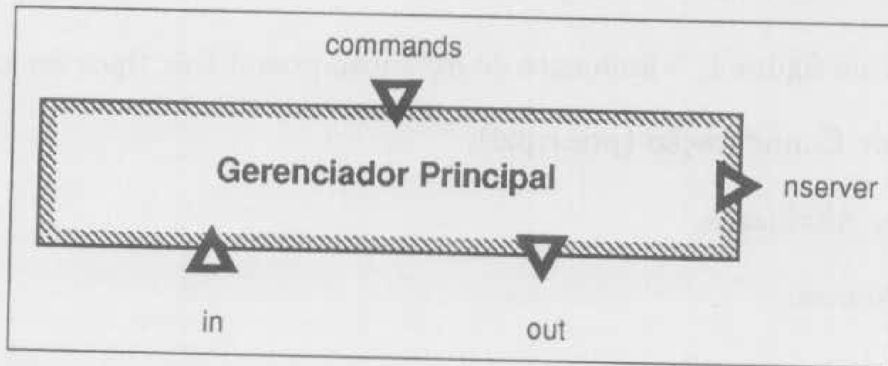

Figura 2: *Interface do Gerenciador Principal*

A inicialização do ambiente de execução é feita criando-se apenas o gerenciador principal. Este, por sua vez, cria na máquina em que reside, o servidor de nomes e uma instância do gerenciador auxiliar. Uma vez criados os módulos iniciais do ambiente, o gerenciador principal passa a esperar por comandos de configuração.

Ao receber um comando de configuração, o gerenciador principal consulta o servidor de nomes para saber em qual máquina aquele comando deve ser executado para então, enviá-lo ao gerenciador auxiliar correspondente.

2.1 O Servidor de Nomes

O servidor de nomes é o responsável por manter informações sobre os módulos da aplicação. Estas informações são de fundamental importância para a execução dos comandos de configuração, e são utilizados pelo gerenciador principal para coordenar esta execução.

O servidor de nomes opera respondendo a consultas e comandos de atualização de dados. Como mostra a figura 3, ele possui apenas uma porta, tipo pedido-resposta orientad à conexão.

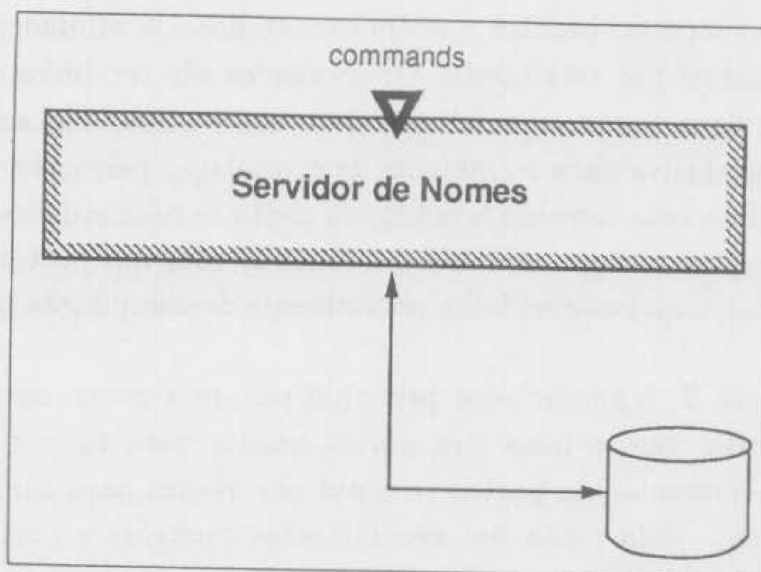

Figura 3: *Interface do Servidor de Nomes*

Por questões de segurança, e devido à importância dos dados que detém, o servidor de nomes utiliza arquivos em disco.

A princípio, o servidor de nomes gerenciaria um arquivo *on-line* contendo todas as informações sobre os módulos da aplicação. O arquivo espelharia o sistema contendo nomes, localização, endereços e estados das portas.

O problema com esta solução é, primeiramente, o grande volume de dados com que os módulos de gerenciamento teriam de manipular, e também as sucessivas atualizações do arquivo necessárias para que o mesmo contenha informações consistentes sobre o estado da aplicação. A manutenção de tal arquivo, além de complexa, tende a atrasar a execução das reconfigurações.

Uma melhor abordagem [KMS87] consiste em se deixar que a aplicação seja sua própria base de dados, ou seja, simplificamos o servidor de nomes para que o mesmo contenha apenas informações sobre a localização dos nós. Isto implica em uma drástica diminuição no volume de dados a serem armazenados e manipulados, pois estas informações, além de pouco volumosas, mudam com pouca freqüência diminuindo sensivelmente o número de atualizações em arquivo. Informações detalhadas sobre cada nó (por exemplo, estado das portas) são obtidas comunicando-se com o próprio nó.

Com a diminuição do volume de dados, torna-se possível que o servidor de nomes guarde um número razoável de informações em memória, o que permite pronta disponibilidade destes dados para o gerenciador principal. O servidor de nomes reserva uma porção de memória para servir de *cache* do arquivo em disco, diminuindo o tempo de resposta ao gerenciador principal e, também, o número de acessos a disco, principalmente nos casos de consulta. Sempre que um dado é buscado no arquivo, uma cópia deste fica guardado na *cache*.

A base de dados é armazenada em um arquivo seqüencial, indexado por uma função *hash* que converte nomes em endereços no arquivo. Este tipo de acesso é mais eficiente que busca seqüencial ou simplesmente indexada.

Os registros na base de dados aqui utilizada, são indexados pelo nome dos processos. Isto é bastante apropriado por que os comandos de configuração possuem os nomes dos processos que tratam.

Basicamente, o servidor de nomes disponibiliza duas operações de consulta e duas de atualização. Uma das consultas, como já dito, consiste em receber o nome de uma máquina e verificar se existem módulos da aplicação naquela máquina. Na outra consulta, o servidor recebe o nome de um processo para determinar em que máquina tal processo se encontra. A maioria dos comandos de configuração recebidos pelo gerenciador principal necessita desta consulta. O servidor de nomes precisa ser atualizado apenas em duas situações, após um processo ser criado, e após ser removido. Isto acontece por que o servidor não guarda informações de estado dos processos.

Como vimos anteriormente, o servidor de nomes guarda poucas informações sobre cada processo rodando sob responsabilidade do ambiente de execução. Vimos também as vantagens qua esta redução no volume de informações traz para a eficiência na execução dos comandos de configuração. No caso particular do servidor de nomes, como conseqüência da redução de atualizações feitas pelo gerenciador principal, tivemos uma simplificação bastante expressiva em seu código. A redução da complexidade do servidor

de nomes possibilitou uma maior concentração de esforços em sua propriedade mais interessante, que é a eficiência em disponibilizar os dados da aplicação para o gerenciador principal.

2.2 Gerenciadores Auxiliares

Os gerenciadores auxiliares são módulos de gerenciamento cujo papel é executar os comandos de configuração nas máquinas onde residem, sob o comando do gerenciador principal. Portanto, um gerenciador auxiliar é instanciado, dinamicamente, em cada máquina onde haja ao menos um módulo da aplicação. A criação dos gerenciadores auxiliares é feita sob demanda pelo gerenciador principal, à medida que novas máquinas vão sendo utilizadas pela aplicação. Cada instância do gerenciador auxiliar gerencia exclusivamente processos locais, abstraindo esta tarefa do gerenciador principal que se preocupa apenas com a coordenação do ambiente.

Como vemos na figura 4, o gerenciador auxiliar possui uma porta de entrada (*commands*) exclusiva para receber comandos do gerenciador principal. Possui também uma porta de entrada (*in*) para receber conexões de processos locais. E uma única porta de saída (*out*) usada para solicitar conexões com processos locais ou com o próprio gerenciador principal, para reportar a ocorrência de falhas ou qualquer evento anormal. Todas as portas suportam apenas comunicações orientadas à conexão.

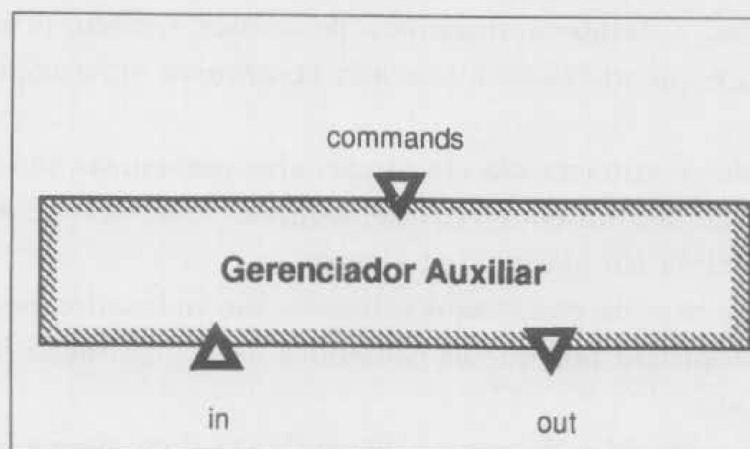

Figura 4: *Interface dos Gerenciadores Auxiliares*

Para executar um comando de configuração, o gerenciador principal o envia ao gerenciador auxiliar correspondente à máquina onde o comando deve ser executado. Para tanto, o gerenciador principal necessita do nome da máquina e do endereço da porta *commands* do gerenciador auxiliar. Estas informações são recuperadas do servidor de nomes. Assim, cada gerenciador auxiliar, ao ser criado, informa o endereço de sua porta *commands* ao gerenciador principal para ser guardada no servidor de nomes.

Entretanto, para simplificar esta tarefa e tornar a criação de gerenciadores remotos mais ágil, adotamos no ambiente de execução o conceito de **endereço conhecido** (*well-known address*) para as portas *commands* dos gerenciadores auxiliares. Logo que é criado, o gerenciador auxiliar abre a porta *commands* com este endereço e assim, este

já fica pronto para servir ao gerenciador principal imediatamente, sem a necessidade de comunicar endereços nem mesmo de passá-los ao servidor de nomes.

Para a execução dos comandos de configuração, cada gerenciador auxiliar guarda, em memória, o nome, *process_id*, e endereço dos processos locais. São basicamente as mesmas informações guardadas pelo servidor de nomes e, além de agililzar a execução de comandos de configuração, esta replicação dos dados é importante para a segurança. Outra vantagem é que o gerenciador principal pode passar os comandos de configuração praticamente da mesma maneira que recebe, com pouco ou nenhum tratamento (e.g., tradução de nomes para endereços), uma vez que os gerenciadores auxiliares já possuem informações suficientes pra executar os comandos independentemente em suas máquinas. O único comando que neessita de algum tratamento antes de ser enviado a um gerenciador auxiliar é o comando `link`, já que o compilador passa ao gerenciador principal apenas os nomes das portas a serem conectadas (uma de saída e uma de entrada). Assim, o gerenciador principal necessita resolver o endereço da porta cujo processo não é local ao gerenciador que vai executar o comando.

Os gerenciadores auxiliares executam os comandos de configuração através de comandos de sistema operacional, para modificar seus estados de execução, ou comunicando-se diretamente com os processos locais para alterar o estado de configuração das portas destes processos.

Quando um novo processo é criado, a primeira coisa feita é o estabelecimento de uma conexão entre este e o gerenciador local onde são passadas as informações a serem guardadas junto ao servidor de nomes. Depois disto, o novo processo fica suspenso aguardando que o gerenciador local faça as devidas conexões de suas portas e o ponha em atividade.

O ambiente de execução garante que qualquer comando executado pelo gerenciador auxiliar seja imediatamente refletido nos módulos da aplicação. No caso de situações onde o pronto atendimento ao comando possa causar inconsistências na aplicação (e.g., desfazer uma conexão com uma transação pendente), o ambiente garante que o comando será executado em um tempo finito e tolerável, já que todas as transações têm um tempo finito para serem executadas.

3 Sistema de Comunicação

O mecanismo que forma a base do sistema de comunicação é o *berkeley sockets*, utilizando o protocolo UDP/IP (datagrama), que é um protocolo não orientado a conexões. O uso deste procotolo tem se mostrado satisfatoriamente eficaz e seguro para ser usado em aplicações distribuídas e de tempo real [SDM84], principalmente quando utilizado em uma rede local padrão Ethernet. A eficiência foi fator decisivo nesta escolha. Datagramas não necessitam de estabelecimento de conexão para serem enviados, nem de mensagens de controle para checar a integridade da conexão que são características comuns em protocolos como o TCP/IP. Outros pontos negativos dos protocolos orientados a conexão são a preocupação de sincronização dos módulos da aplicação nas fases de estabelecimento das conexões e o suporte a comunicações *1-n* e *n-1*. A nível de protocolos de transporte, não existem comunicações desta espécie. O ambiente de execução deve quebrar comuni-

cações multidestino em várias *1-1*. Utilizar TCP/IP neste caso implicaria estabelecer e quebrar várias conexões cada vez que uma mensagem multidestino fosse enviada. Assim, a comunicação entre as portas dos módulos da aplicação é feita utilizando-se o UDP/IP.

Na comunicação entre os módulos de gerenciamento, e entre estes e os módulos da aplicação, foi usado o protocolo TCP/IP. Nestas comunicações, o fator segurança é mais importante que a eficiência. Protocolos orientados à conexão fornecem controle de retransmissão e ordenamento de mensagens. Além disto, a fase de estabelecimento da conexão serve como uma checagem no estado do módulo destino, pois se a conexão for completada com sucesso podemos deduzir que o módulo remoto está em execução normal.

Para obtermos boa eficiência na transmissão de mensagens, não foi utilizado *microkernel*, nem módulos intermediários no sistema de comunicação, como vemos em CONIC[SK87], no ambiente RIO[WL93, SLL94], e no sistema MOSKITO [Bec92, NG90]. Não podemos esquecer que o *microkernel* e os módulos do sistema de comunicação, de qualquer ambiente executando sobre o Unix, são processos sujeitos ao escalonamento do sistema operacional, e o envio/recebimento de mensagens são efetuados apenas quando estes módulos estiverem ativos. A menos que a prioridade dos módulos do ambiente seja aumentada (possível apenas se o usuário possuir a senha da conta *root*, o que geralmente não é o caso) as transmissões são efetuadas apenas durante as fatias de tempo cedidas pelo sistema operacional aos módulos do sistema de comunicação. Isto sem falar das múltiplas cópias que são feitas de uma mensagem até que ela chegue ao seu destino.

No ambiente DisCo, a aplicação roda livre de qualquer interferência por parte do ambiente de execução. Os módulos de gerenciamento do ambiente atuam na inicialização da aplicação e nas suas reconfigurações, ficando inativos o restante do tempo. Os módulos da aplicação são conectados sem qualquer módulo intermediador e as mensagens são transferidas diretamente entre suas portas. Este tipo de sistema de comunicação apresenta uma eficiência bastante elevada para comunicações locais e remotas. Eliminando-se agentes intermediários, elimina-se o excesso de cópias feitas de cada mensagem, diminuindo seu tempo de transmissão.

Uma das funções dos módulos intermediadores existentes no sistema de comunicação de ambientes como CONIC, é auxiliar no estabelecimento e controle das conexões. Isto é possível porque estes módulos centralizam todas as comunicações da aplicação. Além disto, eles facilitam a implementação e manipulação do enfileiramento de mensagens nas portas dos processos. Porém, permitir uma centralização nas comunicações dos módulos implica na queda em eficiência do sistema pois os módulos deste sistema se tornam um gargalo na comunicação.

No ambiente DisCo, não temos o auxílio do *microkernel* ou de qualquer módulo para tratar mensagens ou controlar as comunicações. Em verdade, o sistema de comunicação do ambiente DisCo não é composto por nenhum módulo. Ao invés disto, dotamos os módulos da aplicação com a capacidade de manipulação de suas portas. Assim, há uma parte do sistema de comunicação em cada módulo da aplicação. Por estar diluído na aplicação, o sistema de comunicação, assim implementado, permite controle rápido e eficiente de todas as portas dos módulos sem prejuízos para a transparência na programação. Como não temos módulos intermediários nas conexões para facilitar este controle, atuamos nas conexões através dos próprios módulos da aplicação, sem prejuízos para a transparência

do sistema de comunicação.

Para atingir este objetivo, acopla-se um código extra, em cada módulo da aplicação, que é responsável pelo tratamento das conexões. Este código funciona independente do código escrito pelo programador do módulo, e é totalmente transparente. Ele possui uma porta especial para se comunicar com o gerenciador local (figura 5).

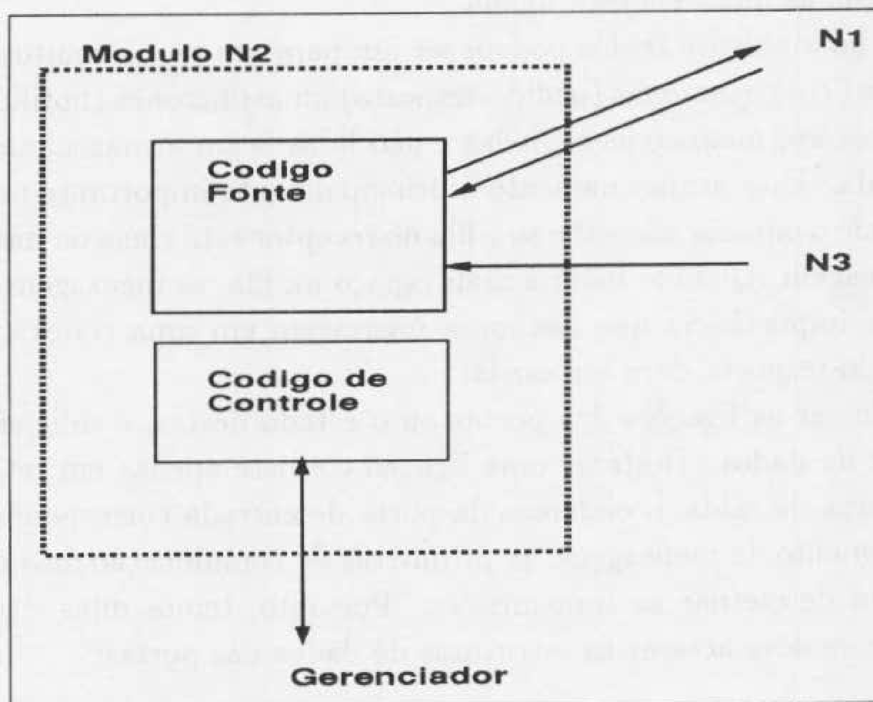

Figura 5: *Estrutura Interna de um Módulo em Execução*

Este código de controle tem a função de gerenciar todo o módulo sob orientação do gerenciador local. Tem controle total sobre as portas e sobre a execução do módulo, podendo inclusive interrompê-la. Pode estabelecer ligações, bloqueá-las ou extingui-las. Mais ainda, garante a prioridade das mensagens vindas do gerenciador local, processando-as tão rapidamente quanto possível. Na próxima seção falaremos mais deste código e de como ele resolve os problemas de controle dos módulos e de suas portas.

3.1 Portas

Como sabemos, as portas dos módulos são descritores para acesso a *sockets*. Para utilizar um *socket*, um processo necessita não apenas de um descritor (como no caso de arquivos), mas também de estruturas (`struct sockaddr_in`, capítulo 3) com informações sobre o seu *socket* e sobre o que está na outra extremidade do canal de comunicação.

Para cada porta declarada pelo usuário, é criada uma estrutura de dados que contém todas as informações sobre a mesma. Obviamente, este trabalho é feito pelo código de controle. A estrutura de dados de qualquer porta tem o seu endereço, seu tipo e estado. No caso de uma porta de entrada, existe uma parte da estrutura que guarda as informações sobre o emissor da última mensagem lida, para o caso de portas pedido-resposta. Ter apenas um campo para esta finalidade não inviabiliza a existência de comunicação *n-1*,

tipo cliente-servidor, pois o servidor lê apenas uma mensagem por vez, faz o tratamento necessário e envia a resposta antes de ler a mensagem seguinte.

As ligações entre os processos, são indicadas quando o endereço da porta de entrada está guardado na estrutura de dados da porta de saída. A estrutura de uma porta de saída pode conter mais de um endereço de porta de entrada, para o caso de comunicações *1-n*. Sua estrutura de dados contém uma lista, alocada dinamicamente, de endereços de portas de entrada às quais ela está ligada.

Conexões no ambiente DisCo podem ser um para um (*1-1*), muitos para um (*n-1*) e um para muitos (*1-n*), síncronas (pedido-resposta) ou assíncronas (notificadas). Em todos os tipos de conexões, mensagens enviadas e não lidas ficam armazenadas em uma fila na porta de entrada. Este armazenamento é principalmente importante para as transações assíncronas, onde o emissor não sabe se a fila no receptor está cheia ou não, antes de enviar uma nova mensagem. Quando não há mais espaço na fila, as mensagens são descartadas. Se é de grande importância que nenhuma mensagem em uma conexão seja perdida, a transação pedido-resposta deve ser usada.

Para modificar as ligações das portas ou o estado destas, é suficiente modificarmos suas estruturas de dados. Desfazer uma ligação consiste apenas em retirar da estrutura de dados da porta de saída, o endereço da porta de entrada correspondente. No caso de envio ou recebimento de mensagens, as primitivas de comunicação devem consultar estas estruturas a fim de efetuar as transmissões. Portanto, temos duas situações em que o código de controle deve acessar as estruturas de dados das portas:

1. Quando o gerenciador local envia um comando que modifica a ligação ou estado de uma porta;

2. Quando é executada uma operação de leitura ou escrita em uma das portas.

Geralmente nestes casos, o código de controle recebe o nome da porta e tem de atualizar ou consultar outras informações a respeito dela. A solução é colocar todas as estrutura de dados das portas em uma lista encadeada. Esta lista é percorrida pelo código de controle sempre que os dados de uma porta forem necessários.

Uma dificuldade desta estratégia consiste exatamente em montar esta lista encadeada porque o usuário declara suas portas como variáveis no programa. O mesmo código de controle é ligado (*linked*) a todos os módulos da aplicação e não tem acesso às portas do módulo de antemão. O código de controle necessita saber quantas e quais são as portas do módulo.

O agente mais indicado para fornecer esta informação é o compilador dos módulos. Quando o compilador detectar uma declaração de porta, ele deve recuperar o nome da variável e passá-la ao código de controle inserindo uma chamada a uma função específica para criar a porta, criar um *socket* e seu descritor, atribuir-lhe um endereço, e por fim, criar e preencher a estrutura de dados. Como a programação dos módulos-tarefa é feita na linguagem C, o ambiente de compilação [dP93] precisa apenas de um pré-processador para fazer este tratamento antes de o módulo ser realmente compilado.

No caso das operações de leitura e escrita, a necessidade de fazer uma busca na lista de estruturas de dados das portas vai depender de que informação a primitiva de

comunicação vai receber. Nas primitivas que vimos no capítulo 5, o usuário passa a variável declarada como **entryport** ou **exitport** para as primitivas de comunicação. Por exemplo:

```
...
exitport out;
char message[32];
...
send message to out;
```

Note que a primitiva **send** recebe a variável **out** declarada como do tipo **exitport**. Tudo vai depender deste tipo. Se desejarmos que a primitiva receba o descritor do *socket* da porta **out**, então **exitport** deve ser inteiro. Entretanto, para evitar a busca na lista de estruturas, e conseqüentemente, aumentar a eficiência das primitivas de comunicação, os tipos **exitport** e **entryport** foram declarados como ponteiros para as estrutura de dados das portas. Assim, ao declarar uma porta, o usuário estará, na verdade, declarando um ponteiro para a estrutura de dados daquela porta, e esta estrutura será colocada na lista. Nas chamadas a primitivas de comunicação, o próprio usuário passa este ponteiro (**out**, no exemplo), eliminando a necessidade de percorrer a lista de estruturas.

4 Suporte a Configuração

Para inicializar o ambiente é necessário apenas executar o gerenciador principal. Vimos que o gerenciador principal cria o servidor de nomes e um gerenciador auxiliar, na máquina em que é inicializado.

Antes de criar estes dois módulos, o gerenciador principal precisa determinar o endereço da *well-known port*, que identificará o sistema e será o endereço das portas de controle dos gerenciadores auxiliares.

O Unix não atribui automaticamente um endereço de porta *internet* com valor maior que 5000. Esta faixa de endereço é deixado para servidores não-privilegiados, desenvolvidos por usuários. Isto fornece um maior grau de certeza que um servidor pode ter um *well-known address* para sua porta, já que qualquer das portas na faixa 1024-5000 pode estar sendo usada por algum cliente. Em geral, os clientes criam suas portas e deixam que o Unix lhes atribua um endereço qualquer, porque eles não necessitam de uma porta com endereço conhecido, precisam apenas de uma porta disponível para acessar um servidor. A faixa de 1024-5000 é utilizada para estas portas de clientes. Assim o gerenciador principal escolhe, aleatoriamente, uma porta com endereço acima de 5000 e testa para saber se está disponível. Se a porta não estiver disponível, testa outro endereço até encontrar um disponível.

Além dos gerenciadores auxiliares, este endereço é utilizado também pelo servidor de nomes para gerar o nome do arquivo onde os dados da aplicação vão ser guardados. O uso de um arquivo para guardar os dados da aplicação aumenta a segurança dos dados

e praticamente torna ilimitado o número de processos permitidos para uma dada aplicação. Em contrapartida, é degradado o tempo de resposta às consultas e comandos do gerenciador principal.

Para rever o nível de eficiência necessário para o gerenciador principal, foram adotadas algumas estratégias. Primeiramente, o arquivo utilizado pelo servidor de nomes é criado no diretório `/tmp`. Este diretório, existente em cada máquina, é destinado ao armazenamento de arquivos temporários. Ao utilizar um diretório no sistema de arquivos da máquina, o servidor de nomes tenta evitar o acesso a arquivos remotos, que pode ser afetado pelo tráfego na rede e pela carga da máquina remota.

Em segundo lugar, o servidor de nomes utiliza um esquema de acesso ao arquivo mais eficiente que busca seqüencial e simplesmente indexada. É a indexação com função *hash*, utilizando subrotinas do `ndbm`(3x). A recuperação de registros em arquivos assim organizados é bastante rápida e direta. A função *hash* encontra o endereço de qualquer registro no arquivo e acessa o próprio registro em um tempo constante, ou seja, de ordem O(1).

Em terceiro lugar, o servidor de nomes guarda, em memória, uma espécie de memória *cache* do arquivo em disco. Nesta *cache* ficam os registros mais recentemente requisitados pelo gerenciador principal. Qualquer mudança nestes registros deve ser imediatamente refletida no arquivo. A política de substituição nesta *cache* é remover o registro "menos recentemente utilizado". A *cache* é utilizada como uma lista duplamente encadeada de registros. O posicionamento do registro na lista reflete seu tempo de utilização. Um registro ao ser utilizado ou incluído passa para o primeiro lugar na lista. Esta troca de posições não envolve de maneira alguma cópia de registros de um lado para outro. Como estão em uma lista duplamente encadeada, uma operação de troca de posições envolve apenas a manipulação de poucos ponteiros de memória. Com as sucessivas trocas de posição, os registros menos utilizados vão ficando no final da lista, tornando trivial o algoritmo de substituição.

Por último, a comunicação do gerenciador principal com o servidor de nomes, que é orientada à conexão, utiliza portas dedicadas em ambos os módulos. Isto permite que a conexão fique ativa durante todo o tempo de execução destes módulos, livrando o tempo que seria perdido com os sucessivos estabelecimentos e quebras de conexões. A comunicação fica, então, segura e rápida. No caso dos gerenciadores auxiliares, este procedimento não é viável porque podem haver muitos deles espalhados pela rede. Além disto, a comunicação entre o gerenciador principal e um dado gerenciador auxiliar não é tão freqüente como a do servidor de nomes.

5 Avaliação de Desempenho

Apesar de os sistemas para atualização dinâmica serem pesquisados desde a década de 70, poucos dados sobre o desempenho de tais sistemas têm sido publicados [SF93].

A falta de publicações neste campo é provavelmente causada pela dificuldade em se caracterizar o desempenho destes sistemas, inviabilizando a coleta de dados significativos, e também porque a maioria dos sistemas são construídos como protótipos apenas para

pesquisa, e nunca são utilizados em aplicações reais.

No caso dos sistemas para atualizações dinâmicas para sistemas distribuídos, uma outra justificativa surge. Ambientes distribuídos são comparados de acordo com várias propriedades além do desempenho. Neste âmbito, o desempenho de maior significância é o do sistema de comunicação, porque é nas comunicações, durante uma execução normal, que as aplicações gastam a maior parte do tempo. Além disto, estratégias adotadas na parte de suporte do ambiente (como o uso de *microkernel* ou *lightweight processes*), refletem diretamente no desempenho do sistema de comunicação.

Testamos o desempenho do ambiente de execução em comparação com os mecanismos mais comuns de comunicação entre processos. O ambiente físico utilizado foram as instalacões do Departamento de Informática da UFPE, que consta de dezenas de estações Sun, estações IBM Risc6000, PC's 486 (rodando LINUX) e terminais X, distribuídas em vários laboratórios e salas do departamento, comunicando-se através de uma rede padrão Ethernet de 10 Mbits/segundo.

Utilizamos um dos laboratórios cuja rede foi mantida com tráfego exclusivo para o experimento, isolando-a da rede externa ao departamento e dos outros laboratórios. Foram mantidas ativas apenas poucas máquinas exclusivamente para o teste: duas estações Sun 4/20 sparc SLC *diskless*, onde os processos foram executados, mais uma estação servidora das SLCs e uma estação servidora de arquivos.

Os processos consistiram de um servidor e um cliente que enviavam e recebiam mensagens (*round trips*), utilizando o protocolo UDP/IP. Nenhum processamento foi feito nos dados transmitidos — nosso interesse era saber o quão rapidamente as mensagens são enviadas. Os programas foram executados usando três tamanhos de mensagens: 32, 512 e 2048 *bytes*. É natural que mensagens maiores requeiram mais tempo para serem transmitidas, mas esta variação é importante para termos uma idéia da influência que o tamanho da mensagem tem no desempenho da comunicação.

Para cada execução, um tamanho de mensagem diferente foi utilizado, e 10.000 mensagens daquele tamanho foram trocadas. Nas medições foram considerados apenas os tempos do *round trip*. Os tempos de criação dos canais de comunicação, e outros procedimentos como cálculo e impressão de resultados, não influiram nos resultados. As amostras de tempos coletadas foram somadas, tiramos média, variância e desvio padrão para cada caso. A tabela 1 mostra os resultados obtidos, em milisegundos, para mensagens de 32 *bytes*.

Sistema	*Média(ms)*	*Desvio P.*	*Variância*
Socket	1.93	0.33	0.11
RPC	3.95	0.62	0.38
PVM	5.70	4.69	22.03
DisCo	2.59	0.74	0.55

Tabela 1: *Medidas de round trip para mensagens com 32 bytes*

Tomando como base o tempo médio dos *sockets* para um *round trip*, vemos que

o ambiente de execução acrescenta um *overhead* no desempenho menor que 30%. Esta percentagem de tempo é utilizada para o tratamento da porta, bloqueio dos sinais do gerenciador local e teste do *status* da porta para então a mensagem ser enviada. As operações são feitas com chamadas a rotinas do sistema operacional e rotinas de acesso à memória. Nas tabelas 2 e 3, vemos os resultados obtidos para mensagens de 512 e 2048 *bytes*.

Sistema	*Média(ms)*	*Desvio P.*	*Variância*
Socket	3.09	0.60	0.36
RPC	18.02	1.01	1.02
PVM	7.33	4.97	24.70
DisCo	3.75	0.71	0.51

Tabela 2: *Medidas de round trip para mensagens com 512 bytes*

Sistema	*Média(ms)*	*Desvio P.*	*Variância*
Socket	6.97	0.68	0.46
RPC	63.48	0.93	0.86
PVM	12.22	4.98	23.95
DisCo	7.43	0.80	0.64

Tabela 3: *Medidas de round trip para mensagens com 2 Kbytes*

Vemos, então, que o *overhead* do ambiente DisCo diminui percentualmente em relação ao desempenho dos *sockets* à medida em que aumentamos o tamanho das mensagens.

Ambientes que utilizam *microkernel* e módulos intermediários no sistema de comunicação chegam a quadruplicar o tempo de transmissão dos mecanismos de comunicação que utilizam. Além disto, nestes ambientes, o *overhead* costuma ser proporcional ao tamanho da mensagem.

No ambiente DisCo, como não são feitas manipulações ou cópias desnecessárias da mensagem, seu *overhead* permance praticamente constante e independente do tamanho da mensagem. O mesmo fato não ocorre com os mecanismos de RPC e PVM. O PVM prevê um empacotamento dos dados e uma cópia da mesma para um *buffer* especial antes de ser enviada. Operações inversas são necessárias depois que uma mensagem é recebida. Por isto, observamos no seu desempenho, uma queda proporcional ao tamanho da mensagem.

No teste utilizamos o PVM com canal direto entre os processos. A comunicação normal no mecanismo de PVM é feita através dos seus *daemons*, presentes em cada máquina. Se usássemos este tipo de comunicação, certamente notaríamos um aumento substancial no *overhead* do mecanismo.

A seguir, vemos um gráfico, na figura 6, onde podemos analisar os dados coletados de uma maneira mais cômoda.

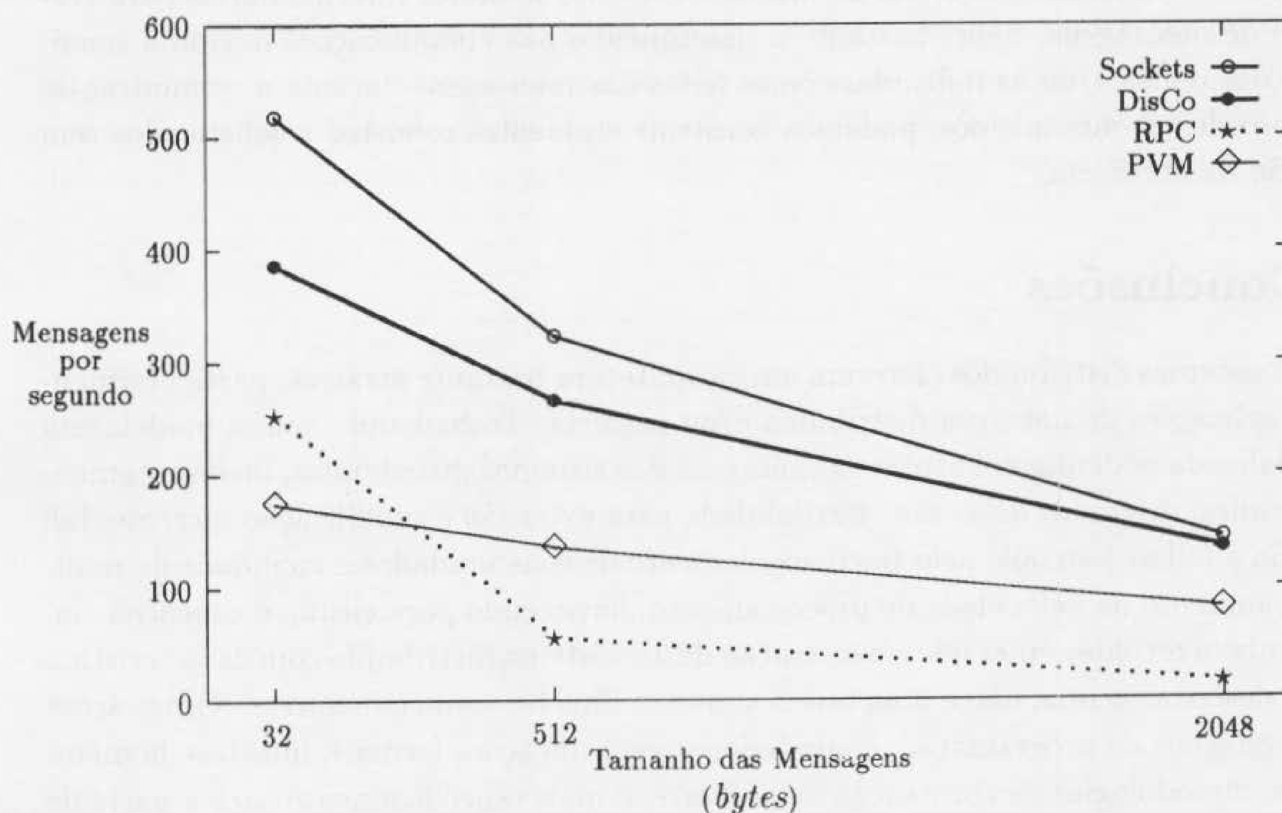

Figura 6: *Comparação do desempenho de vários mecanismos de IPC com o ambiente DisCo*

Um fato importante a ser notado, observando o gráfico, é que, apesar de a quantidade de dados aumentar muito de uma linha para outra, os resultados guardam uma similaridade entre si, não acompanhando, com a mesma proporcionalidade, o aumento no tamanho das mensagens. Iste fato é devido à limitação causada por três fatores:

1. O *overhead* das chamadas a rotinas do sistema operacional e primitivas de comunicação (`read, write, send, receive`);
2. O tempo envolvido em mover os dados entre o processo do usuário e o *kernel* da máquina;
3. O atraso imposto pelas sucessivas trocas de contexto no escalonamento de processos pelo sistema operacional.

Estes fatores indicam que existe um custo fixo, para qualquer forma de comunicação, juntamente com um custo variável que depende da quantidade de dados sendo transferidos. Uma comprovação disto é que a vazão para mensagens de 512 *bytes* não é quatro vezes maior que a vazão para as de 2048 *bytes*. Esta comparação fica ainda mais evidente para valores de 32 e 512 *bytes*.

O *overhead* na comunicação causado pelo ambiente de execução, acrescenta um peso apenas no custo fixo da comunicação, sendo o custo variável igual ao dos *sockets*, que formam sua base de comunicação. Por isto, o desempenho do ambiente de execução acompanha tão bem o desempenho dos *sockets*.

Portanto, notamos que o uso de *microkernel* e/ou módulos intermediários para tratamento de mensagens, reduz bastante o desempenho nas comunicações devido à manipulação dos dados e/ou às múltiplas cópias feitas das mensagens durante a comunicação. Sem o uso destes mecanismos, podemos construir ambientes robustos e sofisticados sem abrir mão da eficiência.

6 Conclusões

Os sistemas distribuídos oferecem uma arquitetura bastante atrativa, particularmente para aplicações de natureza distribuída e/ou paralela. Trabalhando com a modelagem descentralizada podemos confirmar as vantagens dos sistemas distribuídos, incessantemente publicadas. Algumas delas são: flexibilidade para extensão e modificação incremental; tolerância a falhas parciais, pelo fraco acoplamento de suas unidades e facilidade de replicação; e aumento na velocidade de processamento, fornecendo paralelismo e concorrência.

Também reconhecemos que a construção de um sistema distribuído com características como as descritas acima, não é uma tarefa simples. Envolve conhecimento em várias áreas como linguagens de programação, compiladores, especificações formais, interface homem-máquina, metodologias de construção de *software*, e mais especificamente para a parte de suporte, comunicação em rede e seus protocolos, sincronização, e conhecimento aprofundado em sistemas operacionais, incluindo manipulação de memória, temporização (*timing*) e gerenciamento dinâmico de processos.

Neste artigo, descrevemos a construção do ambiente de suporte à execução do ambiente DisCo, proposto para a linguagem de configuração CL. Este ambiente utiliza o paradigma de configuração como base para construção de aplicações. Com o uso deste paradigma, conseguimos separação entre programação e configuração, promovendo **flexibilidade**, ajudando a manter e gerenciar a estrutura das aplicações para possibilitar **reconfigurações dinâmicas**. Sistemas onde as informações de configuração e de processamento estão mescladas na linguagem de programação, e conseqüentemente, nos processos, tornam inviáveis as atualizações dinâmicas.

Executamos com sucesso o objetivo de portar o ambiente DisCo, e consequüentemente a linguagem CL, para uma plataforma distribuída com estações de trabalho em rede. O ambiente de execução foi construído independentemente do ambiente de compilação [dP93] e da interface gráfica [dS94]. Esta separação permitiu uma maior especialização das partes e um aumento na qualidade do serviço prestado por cada uma delas. Para o ambiente de execução esta separação foi particularmente importante pois possibilitou que ele fosse construído de maneira **modular** e **extensível**, para suportar versões diferentes do compilador. Uma vez bem definida a interface, compiladores em desenvolvimento e até mesmo de outras linguagens de configuração podem fazer uso do ambiente de execução.

As estratégias adotadas na construção do ambiente de execução permitiram aliar níveis adequados de segurança e eficiência, tanto no sistema de comunicação quanto no gerenciamento da configuração.

A tarefa de gerenciamento dos módulos distribuídos foi substancialmente facilitada com a utilização de gerenciadores auxiliares em cada máquina. Esta decisão aumentou a

modularidade do sistema, pois cada gerenciador administra apenas recursos e processos locais; simplificou a implementação do gerenciador principal que se preocupa apenas em coordenar o ambiente e a execução dos comandos de configuração; aumentou a **eficiência** na execução destes comandos, pois já existindo um módulo responsável por cada máquina, apenas uma comunicação é necessária para iniciar a execução de um comando. Nenhuma *shell* ou processo é criado para executar um comando remoto. Por fim, a **segurança** do sistema foi garantida pois cada gerenciador auxiliar é o processo "pai" de todos os módulos de sua máquina. No Unix, esta característica é fundamental para a vigilância dos processos, pois o sistema operacional informa qualquer alteração no estado de um processo ao seu processo "pai". Assim, qualquer comportamento anormal da aplicação é imediatamente reportada, permitindo que decisões rápidas sejam tomadas para a restauração da aplicação.

O sistema de comunicação utiliza mecanismos de comunicação eficientes, bastante difundidos e disponíveis publicamente, facilitando a **portabilidade** do ambiente para outras plataformas de *hardware*, com pouca ou nenhuma modificação. A eficiência dos mecanismos utilizados como base para o sistema de comunicação, aliada às estratégias adotadas de construção deste sistema, sem o uso de *microkernels*, módulos intermediários ou *light weight processes*, beneficiou decisivamente a **eficiência na transmissão de mensagens** entre os módulos da aplicação. Esta eficiência foi comprovada pelos testes de avaliação de desempenho realizados.

Na construção do ambiente de execução, pudemos ainda sentir a contribuição das técnicas de especificação formal, no entendimento, desenvolvimento e documentação de sistemas complexos, com uma especificação formal (parcial) do ambiente de execução utilizando a linguagem LOTOS (não incluída neste artigo e descrita em [Ban94]).

Referências

[Ban94] Sílvio Soares Bandeira. Desenvolvimento de um Ambiente de Execução Distribuído para Suporte à Linguagem de Configuração CL. Dissertação de Mestrado, Universidade Federal de Pernambuco, Dezembro 1994.

[Bec92] Thomas Becker. Transparent Service Reconfiguration after Node Failures. *International Workshop on Configurable Systems*, Março 1992.

[CndP93] Paulo Cunha e Virgínia de Paula. CL - Uma Linguagem Orientada a Configuração para Sistemas Distribuídos. 2:475–489, Setembro 1993. Anais do XX Seminário Integrado de Software e Hardware, Florianópolis.

[DK75] DeRemer e H. Kron. Programming in-the-large versus Programming in-the-small. *Proceedings of the Local Area Communications Network Symposium*, Maio 1975.

[dP93] Virgínia Carvalho Carneiro de Paula. Implementação de Linguagens de Configuração para Sistemas Distribuídos. Dissertação de Mestrado, Universidade Federal de Pernambuco, Dezembro 1993.

[dS94] Marcus Venicius Virginio de Sousa. Suporte Gráfico para um Ambiente de Sistemas Distribuídos Orientado à Configuração. Dissertação de Mestrado, Universidade Federal de Pernambuco, Julho 1994.

[Dul90] Naranker Dulay. *A Configuration Language for Distributed Programming.* PhD thesis, Imperial College of Science, Technology and Medicine, University of London, Fevereiro 1990.

[JC92] George Justo e Paulo Cunha. Programming Distributed Systems with Configuration Languages. International Workshop on Configurable Distributed Systems, London, Março 1992.

[JCdP93] G. R. R. Justo, P. R. F. Cunha e Virginia C. C. de Paula. Programação de Sistemas Distribuídos Baseada em Linguagens Orientadas a Configuração. *XIX Conferência Latinoamericana de Informática*, Agosto 1993.

[KMS87] Jeff Kramer, Jeff Magee e Morris Sloman. An Overview of Distributed System Construction Using CONIC. Setembro 1987.

[Mos92] Peter D. Mosses. *Action Semantics.* Cambridge University Press, 1992.

[NG90] J. Nehmer e T. Gauweiler. Design Rationale for the MOSKITO Kernel. *Informatik Fachberichte - Springer Verlag*, 264, 1990.

[SDM84] M. Sloman, J. Dramer e J. Magee. A Flexible Communication Structure for Distributed Embedded Systems. Relatório Técnico DOC 83/11, Department of Computing, Imperial College, Londres, Abril 1984.

[SF93] Mark E. Segal e Ophir Frieder. On-The-Fly Program Modification: Systems For Dynamic Updating. *IEEE Software*, Vol. 10(No. 2), Março 1993.

[SK87] Morris Sloman e Jeff Kramer. *Distributed Systems and Computer Networks.* Prentice/Hall International, 1987.

[SKM85] M. Sloman, J. Kramer e J. Magee. The CONIC toolkit for Building Distrubuted Systems. Maio 1985.

[SLL94] Alexandre Sztajnberg, Orlando Loques e Julius C. B. Leite. Implementação e Testes do Sistema de Comunicação do Ambiente RIO. *XII Simpósio Brasileiro de Redes de Computadores*, Maio 1994.

[WL93] José Alberto Vasi Werner e Orlando Gomes Loques. Ambiente RIO: Metodologia e Suporte para Sistemas Configuráveis. *XX SEMISH da Reunião Anual da Sociedade Brasileira de Computação*, Setembro 1993.