Aspectos de implementação e desempenho de um sistema distribuído "Linda"

Dorgival O. Guedes Neto
Osvaldo S. F. Carvalho
Departamento de Ciência da Computação — UFMG
Caixa Postal 702 — 30161 Belo Horizonte - MG
`dorgival@dcc.ufmg.br, vado@dcc.ufmg.br` *

Resumo

Um sistema distribuído para uma rede de estações de trabalho deve levar em consideração aspectos como facilidades de comunicação da rede, acesso a recursos do núcleo do sistema operacional de cada máquina, etc. Este trabalho apresenta as opções adotadas no desenvolvimento de um núcleo para processamento distribuído em uma rede de estações de trabalho no Departamento de Ciência da Computação da UFMG.

São abordadas as opções de implementação possíveis, detalhes do protocolo de comunicação adotado e alguns resultados dos testes de desempenho do sistema já em operação.

Abstract

A distributed system for workstations networks must consider aspects such as network communication facilities, access to each machine's operating systems kernel resources, etc. This paper presents options adopted during the development of a workstations network based distributed processing kernel in the Computer Science Department of UFMG.

Possible implementation options are described, in conjunction with details of the communication protocol adopted and some results of the performance tests of the working system.

*Este trabalho conta com o apoio da Telebrás — contrato no. 416/91 e do CNPq

1 Introdução

Hoje é cada vez maior a busca de soluções práticas para a obtenção de sistemas paralelos. Um ambiente que vem ganhando atenção nesse sentido são as redes de estações de trabalho, por serem sistemas distribuídos de baixo custo e hoje acessíveis a muitas universidades e empresas nacionais. Tais sistemas se prestam não apenas a estudos e simulações, mas também à utilização efetiva como ambientes distribuídos.

Dentre os vários modelos de sistemas distribuídos de interesse, este trabalho aborda a linguagem **"Linda"**, um modelo que pode ser visto como uma mémoria associativa distribuída para a comunicação e criação de processos. Esta visão fornece um nível de abstração superior a sistemas de trocas de mensagens e/ou memórias compartilhadas, como o sistema `PVM`[GHPW89]. Em relação a outros sistemas baseados no compartilhamento de dados, como `Orca`[BKT92], **"Linda"** oferece uma maior flexibilidade e simplicidade de implementação.

Detalhes da linguagem e alterações propostas ao modelo original podem ser encontradas em um artigo anterior[GC92]. Este trabalho apresenta experiências concretas relativas à implementação já concluída com as soluções adotadas e dados de desempenho já obtidos. Nessa linha, esperamos que ele venha a ser de interesse principalmente para aqueles que venham a se envolver na tarefa de criar aplicações e sistemas de desenvolvimento na área de sistemas distribuídos.

Nas seções a seguir apresenta-se a linguagem **"Linda"** já com as alterações propostas, discute-se as ferramentas disponíveis em uma rede de estações de trabalho para implementação de sistemas distribuídos, a implementação propriamente dita e os resultados de desempenho.

2 "Linda"

"Linda" foi proposta em 1982[GB82] e hoje já existem implementações para diversas arquiteturas, como hipercubos[Luc86], máquinas de memória compartilhada como o Encore Multimax e Sequent Balance, redes de VAX[WL88] e IBM RTs[AB89]. Essas implementações utilizam diversas linguagens base, tais como C (a grande maioria), C++[Lel90], Fortran[CG88], Eiffel[Jel90], Smaltalk[MK88], entre outras.

Uma das principais características da linguagem é a simplicidade de programação. É extremamente fácil representar um conjunto de processos cooperantes e suas interações com base em **"Linda"**, mesmo para profissionais com pouco conhecimento na área de processamento paralelo e distribuído. Outro artigo[GCC] apresenta um programa implementado em conjunto com membros da Escola de Música da UFMG, onde é apresentada uma descrição mais detalhada do processo de desenvolvimento de aplicações no sistema.

A implementação descrita aqui foi alterada em relação à proposta original visando facilitar sua utilização e implementação, dispensando a utilização de um pré-compilador. Como dito anteriormente, a discussão de tais alterações pode ser encontrada em outro artigo[GC92]. No restante deste texto será abordada a versão alterada.

2.1 Elementos básicos

"Linda" se baseia na manipulação de conjuntos ordenados de campos denominados tuplas, tendo cada campo um tipo e valor associado. Completando a tupla destaca-se um campo do tipo sequência de caracteres denominado chave da tupla, utilizado na sua manipulação. Na implementação em linguagem C uma tupla é identificada por um `struct` contendo os campos, e um vetor de `char` de 16 posições para a chave.

A operação básica executada sobre tuplas é o "casamento". Uma tupla casa-se com um padrão (um modelo de tupla) se e somente se a tupla e o padrão possuem a mesma chave, e se todos os campos do padrão forem do mesmo tipo que seus correspondentes na tupla, ou seja, se o padrão e a tupla forem compostos por elementos do mesmo `struct`.

As tuplas são mantidas em espaços de tuplas (ET), sendo que nesta implementação cada espaço tem um tipo de tupla a ele associado, isto é, todas as tuplas depositadas em um dado espaço devem ser de mesmo tipo. Cada aplicação pode então definir tantos ET quantos sejam necessários. Ao ser criado, cada ET recebe uma "chave de acesso", pela qual outros processos da aplicação podem referenciá-lo. A manipulação passa a ser feita de forma semelhante à de arquivos no sistema Unix, com primitivas para criação, abertura, fechamento e remoção de espaços; uma vez criado e aberto, cada ET passa a ser referenciado por um descritor.

O fato de se utilizar espaços de tuplas tipados simplifica a definição da operação de casamento: como o tipo já é automaticamente associado ao espaço de tuplas, uma tupla e um padrão se casam em um ET se e somente se as suas chaves são iguais.

Como citado anteriormente, a manipulação de tuplas através de um campo de chave caracteriza o modelo como uma memória associativa, sendo a unidade endereçável (tupla) de tamanho variável.

2.2 As quatro operações

Uma vez definidas as condições para o casamento de tuplas, as quatro operações que caracterizam **"Linda"** podem ser então definidas. Para isto, deve-se considerar as seguintes declarações no programa em **"C-Linda"**:

```
typedef struct {
  /* ... */                           /* Descricao dos campos da tupla    */
} tuple_t;
int     tsd;                          /* Descritor de um espaco de tuplas */
tuple_t tuple;
int     eval_function( int );

BEGIN_TS_DECL                         /* Definicao de um ET               */
  TS_INIT( tsd, tuple_t, "TS Access Key" );
END_TS_DECL

BEGIN_EVAL_DECL                       /* Definicao de um processo linda   */
  eval_function;
END_EVAL_DECL
```

`out(tsd,"Chave",&tuple)` deposita no espaço de tuplas identificado por tsd uma tupla com chave `"Chave"` e demais campos contidos na variável `tuple`. O processo que a executa não é bloqueado.

`in(tsd,"Acesso",&tuple)` uma tupla identificada por uma chave `"Acesso"` é retirada do espaço de tuplas. Os demais campos da tupla retirada são copiados para a variável `tuple`. Se não há uma tupla disponível com a chave indicada o processo é suspenso até que uma surja, quando o processo é liberado. Se há várias tuplas com a chave especificada, a escolha é não determinística.

`rd(tsd,"Ola'!",&tuple)` semelhante ao `in()`, porém a tupla não é removida do espaço de tuplas após o casamento, mas apenas tem seus campos copiados para a variável `tuple`.

`eval(tsd,"Resultado",eval_function,param)` gera uma "tupla viva" que leva à criação de um novo processo, o qual executará as operações definidas na função `eval_function` com um parâmetro inteiro `param`. Ao término da exeção uma tupla com chave `"Resultado"` deverá ser depositada no ET identificado por `tsd` com um único campo (inteiro) contendo o valor de retorno da função. O processo pai, que executou o `eval()`, prossegue sem esperar por seus filhos.

A operação de `eval()` é sem dúvida a mais complexa, e será abordada com mais detalhes oportunamente.

Com base nestas operações, pode-se verificar que o modelo permite acesso concorrente a dados para leitura, porém não há nenhuma operação que permita a alteração de dados enquanto no espaço de tuplas. Uma alteração de uma tupla só é possível através de sua remoção do espaço de tuplas e sua substituição por outra com novo valor.

3 Ferramentas disponíveis para a implementação

Tendo em vista o problema proposto de se implementar **"Linda"** sobre uma rede de estações de trabalho, é necessária uma avaliação das ferramentas disponíveis e opções para o desenvolvimento de sistemas distribuídos em tal ambiente:

3.1 Protocolo de transporte

As duas opções consideradas para o protocolo de transporte a ser adotado foram o desenvolvimento de um protocolo específico e a utilização do conjunto TCP/IP.

A primeira poderia oferecer um maior desempenho, porém a um custo de implementação muito grande. A segunda oferece uma solução mais simples, já com recursos para a utilização de diversos meios físicos sobre os quais TCP/IP[Com88] já foi implementado. Para a implementação de um novo protocolo seria utilizado o recurso de programação por STREAMS oferecido pelos sistemas baseados no UNIX[Sun90b]. Na implementação em questão optou-se pelo conjunto TCP/IP.

Em qualquer caso, duas características são desejáveis: recursos para execução de "broadcast" e para comunicação confiável. No TCP/IP, essas características são oferecidas pelos protocolos UDP e TCP respectivamente.

3.2 Interface de programação

São os três os modelos de interface de programação disponíveis em uma rede de estações de trabalho para a comunicação entre processos[Sun90a]: "Sockets", originários do UNIX desenvolvido em Berkeley, TLI, oferecida no UNIX System V, e RPC, desenvolvido originalmente para as estações de trabalho Sun. Apresentam como vantagens, respectivamente: larga utilização, interface mais elegante (OSI) e simplicidade. Como desvantagem: complexidade, suporte ainda restrito e necessidade de adoção do modelo cliente-servidor.

Optou-se pela utilização de "sockets", uma vez que as implementações de TLI em muitos casos ainda não são completas, como no caso das estações Sun, onde só o protocolo TCP é acessível com TLI até o momento.

4 Implementação do núcleo

As implementações de **"Linda"** usualmente podem ser identificadas pelo método de armazenamento de tuplas nos nodos: com replicação total, com assinalamento local, ou com assinalamento a um nodo determinado em função dos campos da tupla. O primeiro necessita de um mecanismo de "broadcast" confiável e um protocolo complexo para retirada de tuplas. O último consiste basicamente em uma função de "hash" para se definir o nodo, sendo utilizado em implementações para hipercubos[Luc86].

Para a implementação em uma rede TCP/IP, os seguintes pontos devem ser considerados:

- O "broadcast" é implementado utilizando-se os recursos da rede física disponível (ethernet, token ring, etc.). Tal "broadcast" é restrito entretanto a um enlace da rede, ou seja, um barramento ethernet ou um anel.
- O recurso de "broadcast" é restrito ao protocolo UDP, o qual oferece um serviço de datagrama, logo sem garantia de entrega a todos os nodos.
- O protocolo "confiável", isto é, com correção de erros e garantia de entrega é o TCP, que possui um certo "overhead" para o estabelecimento e liberação de conexões.

Uma vez feitas essas considerações foi definido o algoritmo a ser adotado, que é apresentado a seguir.

4.1 O algoritmo básico

O algoritmo adotado foi o de deposição de tuplas no local de sua geração, chamado também **"local `out()`, broadcasted `in()`"**. Para a implementação, cada máquina do sistema executa um processo do núcleo **"Linda"**.

Como citado anteriormente, optou-se pela utilização de "sockets" como a interface de programação, devido principalmente à sua maior disponibilidade. Como a primeira implementação seria restrita a uma rede de estações Sun não se utilizou nenhum modelo de representação de dados independente de arquitetura.

4.1.1 Conexão entre aplicação e núcleo

Como núcleo e aplicação executam em processos Unix separados, torna-se necessário que se crie um meio de comunicação entre eles. Na implementação corrente optou-se pela utilização de uma conexão TCP. Por se tratar de uma conexão em uma mesma máquina a rede não é utilizada. A comunicação é feita pelo núcleo do sistema operacional, o que reduz seu custo.

Apenas um pequeno número de processos de aplicação é esperado em cada nodo, uma vez que vários processos em um nodo estariam concorrendo pelos mesmos recursos de máquina e se prejudicando mutuamente. Assim sendo, não representa um custo muito grande para o núcleo manter uma conexão TCP permanente para cada processo de aplicação local. Como a mesma conexão é mantida durante toda a vida do processo de aplicação, os custos de estabelecimento e término da ligação podem ser desconsiderados.

Conexão: O cliente inicia a conexão e se identifica. Essa informação é utilizada pelo núcleo para distinguir um cliente (aplicação) de um parceiro (conexão do processo do núcleo em outro nodo que será descrita posteriormente).

`out()`: Na execução de um `out()` a aplicação envia ao núcleo a tupla gerada e prossegue imediatamente.

`in()` ou `rd()`: A aplicação envia ao núcleo a identificação da tupla desejada e se bloqueia à espera da resposta. Quando o núcleo encontrar uma tupla satisfatória ele a entrega à aplicação, e aguarda por uma nova mensagem indicando se a tupla deve ser removida do espaço de tuplas (`in()`) ou não (`rd()`). Só após fornecer essa informação o processo da aplicação é liberado para continuar.

Desconexão: Uma aplicação que termine de forma normal envia uma última mensagem para permitir que o núcleo proceda à liberação de recursos que tenham sido alocados para a mesma.

4.1.2 Comunicação entre processos componentes do núcleo

O núcleo trabalha basicamente em resposta a duas operações da aplicacão: requisição e deposição de tuplas. Em ambos os casos, o comportamento do núcleo pode seguir três padrões:

1. Caso o núcleo receba uma tupla depositada por um cliente e não haja nenhuma requisição que esta nova tupla satisfaça, a tupla é simplesmente armazenada localmente no núcleo e nenhuma outra operação é desencadeada.

2. Caso o núcleo receba uma tupla de um cliente e verifique que um outro cliente no mesmo nodo se encontra bloqueado à espera de uma tupla daquele tipo, a tupla é imediatamente entregue. De forma análoga, se o núcleo recebe de um cliente uma requisição que pode ser servida por uma tupla armazenada localmente, a entrega é imediata.

3. Caso o núcleo receba uma requisição que não possa ser servida prontamente, ou receba para armazenamento uma tupla que satisfaça a uma requisição de um cliente em outro nodo, torna-se necessária a utilização do protocolo entre nodos.

Os eventos ocorridos para o caso de uma requisição que não possa ser servida localmente são apresentados na figura 1.

1. O processo cliente envia ao processo do núcleo no mesmo nodo a requisição para um dado tipo de tupla.

2. O processo do núcleo local, ao verificar que não possui uma tupla satisfatória executa um "broadcast" (UDP) de uma consulta para todos os nodos indicando o padrão da tupla desejada.

3. Todos os demais nodos verificam em suas tuplas locais se alguma delas satisfaz ao padrão indicado. Caso encontrem, enviam uma confirmação (UDP) ao parceiro que iniciou a consulta e armazenam o pedido, marcando sua resposta. Caso contrário, a consulta é armazenada como pendente.

4. O processo do núcleo que iniciou a consulta seleciona como parceiro o nodo que entrega a primeira resposta e descarta as demais. Um pedido direto (UDP) é enviado ao nodo selecionado.

5. O processo do núcleo no nodo selecionado verifica se ainda possui uma tupla satisfatória (que pode, por exemplo, ter sido entregue a um outro cliente local). Em caso afirmativo, uma conexão TCP é aberta entre os dois nodos, e o parceiro escolhido entrega a tupla.

6. Caso contrário, uma mensagem UDP indica a negação do pedido, e o nodo requisitante reinicia o processo.

Até o momento da entrega da tupla, não há em nenhum momento um processo de "reserva" de tuplas, isto é, o fato de um nodo responder a uma consulta não impede que ele responda a consultas semelhantes de outros nodos. Tal procedimento criaria tuplas bloqueadas em todos os nodos que respondessem e um mecanismo seria necessário para remover esse bloqueio de todos os nodos que não o escolhido para fornecer a tuplas.

Esse relaxamento das restrições considera ainda que várias tuplas possam existir em um nodo que atendam a uma ou mais consultas. O não bloqueio permite que toda e qualquer dessas tuplas seja tratada em paralelo. Não necessariamente a tupla encontrada pelo parceiro para responder à consulta será a mesma usada na entrega direta.

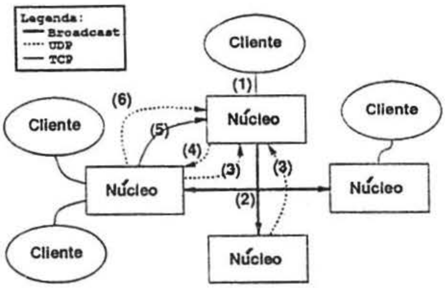

Figura 1: Eventos durante uma consulta/requisição

Esse fato tem outro efeito além do já descrito anteriormente: pode ocorrer que um cliente local deposite uma tupla que satisfaça à requisição de outro cliente local após o núcleo ter iniciado a consulta a outros nodos. Nesse caso, a tupla é entregue ao requisitante, e a requisição cancelada apenas localmente. Quanto aos outros nodos, duas coisas podem ocorrer:

- Se a requisição é cancelada antes do recebimento de qualquer confirmação, basta ao núcleo verificar que não há mais requisição que corresponda a uma dada confirmação. Com isso, nenhum pedido direto será efetuado e os demais nodos prosseguirão sua execução normalmente, uma vez que o envio de confirmação não implica em alterações de estado do processo do núcleo.

- Se a requisição é cancelada após o envio do pedido direto e o pedido for negado a operação termina, uma vez que não há mais a requisição para ser reiniciada. Caso o parceiro venha a entregar a tupla requerida, a entrega é feita ao núcleo, que simplesmente verifica se há um pedido pendente que possa ser servido. Se não houver, a tupla permanece armazenada no novo nodo, havendo apenas uma mudança de localização.

O armazenamento das consultas pelos outros nodos tem por função contornar o problema que pode ocorrer de nenhum nodo possuir uma tupla satisfatória no momento da consulta. Todos os nodos então manteriam o pedido marcado como pendente, e cada nova tupla recebida seria comparada com a consulta em busca de um "casamento". Quando isso acontecesse em pelo menos um nodo o processo continuaria.

4.2 Temporizações

- Uma análise cuidadosa do algoritmo descrito revela, se não uma falha operacional, pelo menos um problema de sobrecarga desnecessária: os nodos não selecionados para a consulta direta não são informados pelo processo originador da remoção da requisição. Além do espaço de armazenamento gasto para armazenar uma consulta não mais válida, há também o gasto adicional de uma confirmação desnecessária.

- Como indicado na descrição do protocolo básico, toda a comunicação que antecede a entrega da tupla é feita utilizando-se o protocolo UDP, mais rápido, porém por ser um serviço do tipo datagrama sem conexão, não confiável. Mensagens podem ser perdidas — e implementações iniciais confirmaram que algumas realmente o são.

Esses dois problemas levam à necessidade de se implementar um mecanismo de temporização nos processos do núcleo, a fim de garantir uma operação sem mensagens desnecessárias nem perdidas.

O problema de expiração de consultas é relativamente trivial. Basta que se defina um tempo de validade para as consultas armazenadas. Expirado esse período, simplesmente remove-se a consulta. Deve-se então atentar para o caso extremo em que todos os nodos temporizam e removem uma consulta antes que uma tupla satisfatória surja. Para evitar que o processo originador espere indefinidamente por uma resposta que não será mais possível, deve haver uma temporização também para o processo de consulta: caso nenhuma confirmação seja recebida após um determinado período — quando por temporização todos os nodos já terão removido a consulta inicial — o protocolo de consulta deve ser reiniciado.

O problema de perda de mensagens pode parecer mais complexo à primeira vista. Uma solução possível sem uma análise mais detalhada do sistema é sem dúvida a implantação de mecanismos de temporização a nível de mensagens isoladas e remoção de mensagens duplicadas. Um mecanismo desse tipo pode ser encontrado em [Ste90]. No caso em questão porém esse mecanismo apresenta um grande inconveniente: inviabiliza a utilização de "broadcast", uma vez que seria necessário uma confirmação para cada nodo da rede, o que aumentaria enormemente o tráfego.

Nesse caso a solução para esse problema pode ser encontrada no método adotado para contornar o problema anterior de expiração de consultas: utiliza-se não a temporização no nível mais baixo, de mensagens individuais, mas sim a nível de consultas. Com isso, caso uma mensagem fundamental se perca, após um certo período sem respostas o nodo originador da consulta pode reiniciá-la. Não há problemas se isso terminar por duplicar a consulta em alguns nodos. Como a transferência de tuplas entre nodos só se dá após a escolha de um único nodo, as mensagens repetidas podem gerar um pequeno tráfego adicional, porém inócuo.

4.3 "Gateways"

Uma rede local de estações de trabalho como a proposta usualmente não se limita a uma única sub-rede (um barramento ethernet ou anel "token-ring") porém o mecanismo de "broadcast" não transpõe tais barreiras. Para superar tal limitação o processo do núcleo implementa um "gateway" de aplicação, fazendo com que o protocolo opere em redes de topologia irregular como na figura 2.

Para isso, processos do núcleo disparados em máquinas que devam agir como "gateways" identificam todas as interfaces daquela máquina. Cada mensagem "broadcast" recebida por uma destas interfaces é retransmitida em "broadcast" pelas demais interfaces.

Os processos do núcleo que operam como "gateway" identificam como especiais os "broadcasts" originados localmente, a fim de evitar uma repetição infinita do processo. Esse esquema impõe como restrição necessária para o seu funcionamento que as sub-redes componentes do sistema não formem laços, o que é usualmente verdade para redes TCP/IP.

5 Manipulação de vários espaços

A definição original de **"Linda"** previa apenas um espaço de tuplas global. Para aumentar o poder de expressão do sistema optou-se por definir um modelo de múltiplos espaços.

5.1 Ambiente de uma aplicação

Um sistema operando com **"Linda"** pode conter várias aplicações independentes, cada qual com vários processos em execução. Torna-se desejável então que se crie um ambiente em que todos

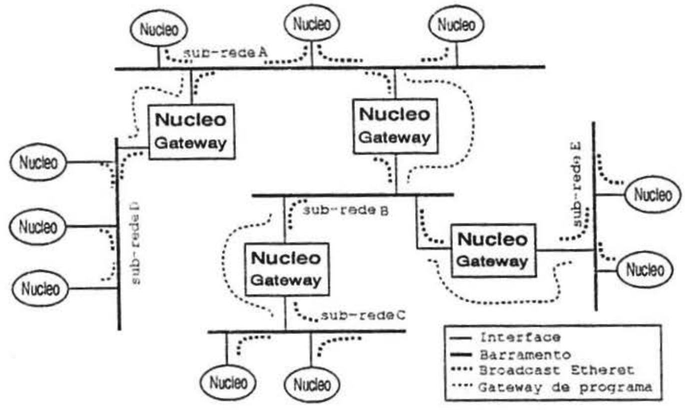

Figura 2: Estrutura de uma rede local de topologia irregular

os processos de uma aplicação tenham condições de identificar e utilizar os mesmos espaços, enquanto o acesso a espaços de outras aplicações seja completamente bloqueado.

Para isso, o identificador de espaços utilizado pela aplicação não pode ser uma referência direta ao índice de um ET no núcleo. Caso isso ocorresse, duas aplicações que utilizassem o ET de índice 1 por exemplo não poderiam ser executadas no mesmo instante. Isso é resolvido utilizando-se um nível de indireção entre índices de aplicação e índices do núcleo, de forma semelhante à tabela de descritores de arquivos de um processo UNIX[Bac86].

Cada aplicação aloca inicialmente um ET para operar como ET base da aplicação, reconhecido por todos os processos através do descritor 0. Os demais ET a serem utilizados são explicitamente declarados no início do programa **"Linda"**:

```
BEGIN_TS_DECL
  TS_INIT( tsd1, tuple_t1, "TS Access Key 1" );
  TS_INIT( tsd2, tuple_t2, "TS Access Key 2" );
  /* ... */
END_TS_DECL
```

Nesse caso, `tsdi` é a variável que irá conter o índice do descritor na tabela de referência da aplicação, a qual deverá ser utilizada sempre para identificar o espaço; `tuple_ti` é o tipo das tuplas a serem manipuladas, e `"TS Access Key i"` uma chave de acesso para o espaço. Essas chaves de acesso são na verdade chaves para tuplas de controle que são depositadas no espaço base, contendo o índice real de cada espaço. A figura 3 ilustra essa situação.

Com isso, cada aplicação só tem acesso aos espaços por ela explicitamente definidos, evitando acessos indevidos. Uma operação com um descritor não inicializado representa um erro de execução da aplicação.

6 Implementação de eval( )

Usualmente `eval()` não é implementado com base nas demais primitivas, uma vez que o ganho em desempenho de uma implementação direta tem se mostrado pouco significativo[Car87].

Utilizando-se as primitivas `out()` e `in()` a tarefa é bastante simples, com base em um "espaço de tuplas vivas". A operação

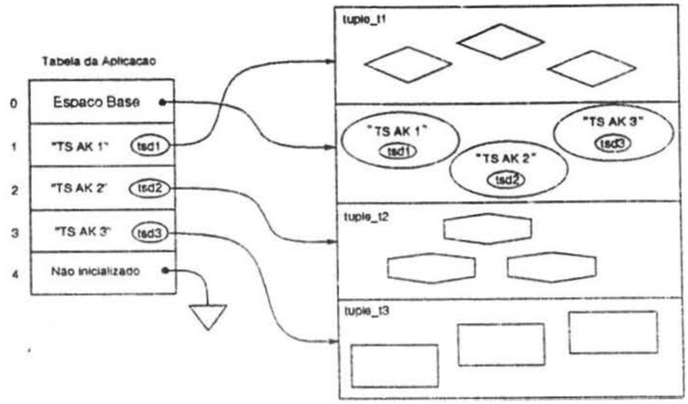

Figura 3: Estrutura de referências para espaços de tuplas

```
eval(atsd,"RESULT KEY",func,param)
```

onde atsd é o descritor do ET onde deseja-se que a função `func()` deposite uma tupla com chave `"RESULT KEY"` com um campo inteiro contendo seu valor de retorno em vista do parâmetro `param` pode ser implementada da seguinte forma:

O "espaço de tuplas vivas" é usado para depositar tuplas que decrevam a função a ser executada. Essa descrição deve englobar:

- A função a ser executada;
- seu parâmetro de entrada;
- o espaço de tuplas em que o resultado deve ser depositado e
- a chave a ser usada para a tupla de resultado.

Ao iniciar a aplicação, o usuário indica todas as máquinas que comporão a aplicação, isto é, as máquinas onde um `eval()` poderá ser executado. Nesses nodos são disparados processos especiais (servidores de `eval()`) que executam um `in()` para as tuplas "vivas".

Uma vez retornando desse `in()`, o servidor de `eval()` executa a função indicada com o parâmetro dado. Em seguida, executa um `out()` com o valor de retorno da função no ET descrito na tupla "viva" com a chave correspondente. O cuidado a se tomar é que o identificador do ET utilizado deve ser o índice na tabela do núcleo. O comportamento do processo que executa o `in()` para tuplas vivas pode ser de duas formas:

- O processo aguarda até o final da execução da função antes de tentar executar um novo `in()`, ou seja, o processo só serve uma tupla viva por vez. Nesse caso, o número máximo de processos da aplicação é definido pelo número de "servidores de `eval()`" disponíveis.
- O processo executa um `fork()` para processar a função e retorna imediatamente ao `in()` para tuplas vivas. Essa solução apresenta como inconveniente que um processo que por algum motivo tenha algum privilégio para executar `in()`'s (por exemplo, por possuir uma conexão mais rápida ao nodo gerador de tuplas vivas) pode vir a executar simultaneamente vários pedidos de `eval()`, os quais poderiam se beneficiar de uma distribuição por mais máquinas.

Uma melhoria poderia ser feita utilizando-se um sistema de determinação dinâmica da carga nas várias máquinas da rede para se determinar quando seria interessante a um nodo processar mais que um pedido concorrentemente. Tal solução envolve, basicamente, um esquema de escalonamento dinâmico de processos em sistemas distribuídos.

Na implementação presente optou-se pelo esquema de alocação estática por sua simplicidade. O programador especifica ao iniciar a aplicação a localização de seus servidores de `eval()`, podendo mesmo optar por utilizar mais de um servidor em um mesmo nodo.

7 Desempenho

A seguir apresentamos alguns dados relativos aos custos de comunicação envolvidos nas operações básicas do sistema. Não se tratam de dados de aplicações reais, mas sim de programas de teste especialmente desenvolvidos a fim de se observar uma dada característica do núcleo.

Foram utilizadas nas medições estações SparcStation2 disponíveis na rede local do Departamento de Ciência da Computação da UFMG. Nos testes envolvendo dois processos foram consideradas três situações, ilustradas na figura 4:

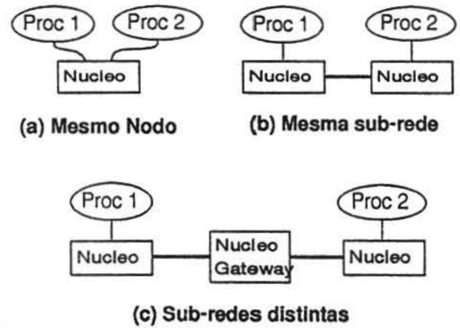

Figura 4: Distribuições consideradas

a: Os dois processos executando no mesmo nodo. Nesse caso, não há atrasos devido ao protocolo de comunicação entre nodos;

b: os dois processos executando em nodos diferentes na mesma sub-rede. Passa a ser relevante o protocolo de interconexão, sem haver um "gateway" de aplicação envolvido;

c: os dois processos executando em nodos diferentes em sub-redes diferentes, envolvendo um "gateway" na comunicação.

7.1 Geração e retirada de tuplas

Para se determinar o tempo das operações de `out()` e `in()` isoladas, foi utilizado o programa a seguir. As tuplas foram definidas como um vetor de inteiros a fim de se estudar o desempenho para vários tamanhos de tuplas.

```
/******* Programa principal *******/
typedef struct { int vector[ TUPLESIZE / sizeof(int) ]; } tuple_t;
BEGIN_TS_DECL
  TS_INIT( tsd, "TS key", tuple_t )
END_TS_DECL
int     tsd;
tuple_t tuple;
lmain() {
  scatter(MAXCOUNT);
  eval( tsd, "Some key", gather, MAXCOUNT );
}
```

```
/******** scatter ************
 * gera tuplas continuamente *
******************************/
scatter(maxcount)
  int maxcount;
{
  for (i=0;i<maxcount;i++)
    out( tsd, "A key", &tuple );
}
```

```
/******** gather ************
 * remove as tuplas geradas *
*****************************/
gather(maxcount)
  int maxcount;
{
  for (i=0;i<maxcount;i++)
    ( tsd, "A key", &tuple );
}
```

Observe-se pelo programa principal que no momento em que o processo para remoção das tuplas é disparado todas elas já foram geradas. Não há assim nenhum atraso devido à espera da geração das tuplas. As constantes MAXCOUNT e TUPLESIZE foram variadas ao longo das medições.

Os gráficos da figura 5 apresentam os resultados para as três configurações para alguns tamanhos de tuplas. As medições foram feitas medindo-se o tempo de execução do `for` de cada função e dividindo-se pelo valor de `maxcount`.

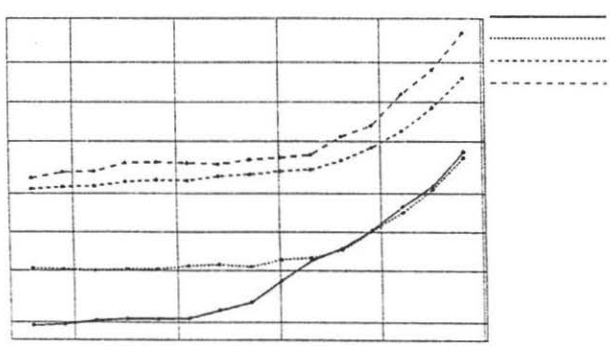

Figura 5: Resultados para a aplicação scatter/gather

Pode-se observar pelo gráfico alguns fatores interessantes sobre o comportamento do núcleo:

- O processamento de um `out()` sem se considerar o processo de transferência da tupla propriamente dita é de aproximadamente 2 ms, enquanto um um `in()` consome 5 ms. Esses tempos podem ser vistos como o tempo de inserção de um novo dado no primeiro caso e de busca da informação no segundo.

- O tempo de execução de um `out()` é dominado pelos custos de processamento para tuplas até aproximadamente 128 bytes, começando a crescer a partir de então em função do tamanho da tupla.
- O tempo de execução de um `in()` por sua vez tem por limite de mudança de comportamento o tamanho de 2048 bytes. Para tuplas abaixo desse valor os tempos do protocolo de aquisição da tupla dominam.
- O "gateway" de aplicação tem por efeito incluir um atraso constante para tuplas até 2048 bytes. Até esse valor o principal custo envolvido na operação é basicamente a retransmissão dos sinais de "broadcast" envolvidos, criando um atraso de aproximadamente 15 ms. A partir de então o atraso cresce, uma vez que se torna sensível o fato da conexão TCP de transferência da tupla passar também pelo gateway, nesse caso no nível do protocolo de roteamento de pacotes da rede.
- Pode-se afirmar que o tempo envolvido no protocolo de localização e requisição de tuplas consome aproximadamente 3 ms no melhor caso, isto é, da tupla já disponível e sem outros pedidos simultâneos.

7.2 Comunicação completa

No caso anterior os dois processos executam sem sincronização. O programa a seguir permite que se avalie o caso inverso, isto é, dois processos que executam com sincronização passo a passo. Com isso pode-se medir o desempenho para um par `out()+in()` completo.

```
/******* Programa principal *******/
 typedef struct { int vector[ TUPLESIZE / sizeof(int) ]; } tuple_t;
 BEGIN_TS_DECL
   TS_INIT( tsd, "PingPong", tuple_t )
 END_TS_DECL
 int     tsd;
 tuple_t tuple;
 lmain() {
   eval( tsd, "Ping Prog", ping, MAXCOUNT );
   eval( tsd, "Pong Prog", pong, MAXCOUNT );
 }
```

```
/*********** ping ************/
 ping(maxcount)
   int maxcount;
 {
   for (i=0;i<maxcount;i++)
   {
     out( tsd, "Ping", &tuple );
     in(  tsd, "Pong", &tuple );
   }
 }
```

```
/******** pong ************/
 pong(maxcount)
   int maxcount;
 {
   for (i=0;i<maxcount;i++)
   {
     in ( tsd, "Ping", &tuple );
     out( tsd, "Pong", &tuple );
   }
 }
```

O gráficos da figura 6 apresentam os resultados para as três configurações para alguns tamanhos de tuplas. As medições foram feitas medindo-se o tempo de execução do `for` da função `ping()` e dividindo-o por `2*maxcount` para se levar em conta os dois pares `in()+out()`.

Muitos dos fatores observados no caso anterior continuam verdadeiros, como o custo de retransmissão pelo "gateway", que se mantém na faixa de 15 ms. Alguns novos dados devem ser observados:

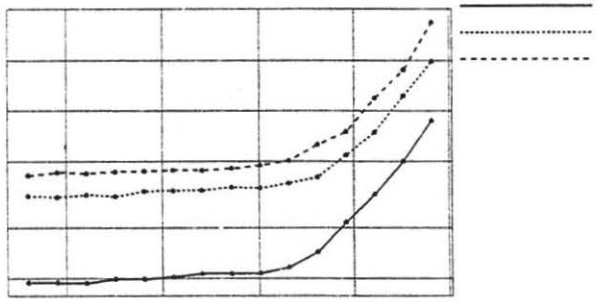

Figura 6: Tempos para a aplicação pingpong

- O tempo gasto a partir de um out( ) até a remoção da tupla por um in( ) se mantém aproximadamente constante para tuplas de até 1024 bytes, quando começa a crescer devido a problemas de bufferização, escalonamento, etc.
- O custo para processamento de um par out( )+in( ) em um mesmo nodo é da ordem de 10 ms, crescendo para 30 ms para hosts em uma mesma sub-rede e 45 ms para hosts em sub-redes vizinhas.

Os dados obtidos levam a algumas observações importantes: apesar do tempo para um out( ) ser constante apenas para tuplas até aproximadamente 128 bytes esse dado é de pouca relevância no tempo final de uma comunicação, uma vez que uma tupla gerada necessitará de um in( ) para ser usada em algum momento, sendo os tempos de comunicação então fortemente dominados pela operação de in( ).

O patamar aproximadamente constante até a tamanhos próximos a 1024 bytes torna mais visível o problema de granularidade. Como seria de se esperar para as características de uma rede local, aplicações de grão muito fino (usualmente com muitas mensagens curtas em relação aos tempos de processamento entre mensagens) tendem a não se beneficiar do aumento do número de processos em tal sistema. O tamanho de 1024 bytes torna-se um ponto de referência na definição dos padrões de comunicação a serem utilizados no desenvolvimento de uma aplicação.

Vale notar que, apesar de servir como referência, tal valor não deve ser considerado como único fator na determinação dos sistemas. Outros fatores tais como taxa de processamento em função da taxa de comunicação e volume de cálculos em função do tamanho das mensagens devem ser considerados.

8 Conclusões

O trabalho de implementação de um sistema distribuído em uma rede de estações de trabalho depende de uma análise cuidadosa das ferramentas disponíveis para o trabalho. No caso em questão optou-se por utilizar a interface de "sockets", oferecida pelo sistema operacional UNIX comumente encontrado naquelas estações, criando um padrão de comunicação que utiliza os recursos de "broadcast" e comunicação rápida do protocolo UDP para mensagens de controle e a comunicação com confirmação do protocolo TCP para a entrega dos dados propriamente ditos.

O sistema apresentou resultados satisfatórios nos primeiros testes de desempenho do módulo de comunicação, apresentando tempos mínimos da ordem de 10 ms para uma comunicação completa entre processos. Os dados confirmam o fato da implementação se adaptar a aplicações de grão grosso, uma vez que há pouca redução dos tempos de comunicação para mensagens menores que 1024 bytes.

Um gerador de conjuntos de mandelbrot já foi implementado, apresentando bons resultados de "speed-up" nas primeiras observações. Novas aplicações se encontram em estudos, bem como continuações para o trabalho de desenvolvimento do núcleo, envolvendo a criação de um sistema de desenvolvimento e depuração, elementos para operação em sistemas heterogêneos e recursos para tolerância a falhas de processos e máquinas envolvidas na aplicação.

Referências

[AB89] Mauricio Arango and Donald Berndt. Tsnet: A linda implementation for networks of unix-based computers. Technical report, Yale University, August 1989.

[Bac86] Maurice J. Bach. *The Design of the Unix Operating System*. Prentice-Hall, 1986.

[BKT92] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[Car87] Nicholas Carriero. *Implementing Tuple Spaces in Linda*. PhD thesis, Yale University, 1987.

[CG88] Nicholas Carriero and David Gelernter. Applications experience with linda. In *Proceddings Symposium on Parallel Programming*, pages 173–187. ACM, July 1988.

[Com88] Douglas Comer. *Internetworking with TCP/IP: Principles, Protocols and Architecture*. Prentice-Hall, 1988.

[GB82] D. Gelernter and A. Bernstein. Distributed communication via global buffer. In *Proceedings Symposium on Principles of Distributed Computing*, pages 10–18. ACM, August 1982.

[GC92] Dorgival O. Guedes and Osvaldo S. F. Carvalho. Um núcleo "linda" para o desenvolvimento de aplicações distribuídas em uma rede unix. In *Anais do X Simpósio Brasileiro de Redes de Computadores*, pages 574–585. SBC, April 1992.

[GCC] Dorgival O. Guedes, Antônio Gilberto M. Carvalho, and Osvaldo S. F. Carvalho. Experiências de programação em "linda": transformações de espectro. Trabalho submetido para o IV SBAC-PAD.

[GHPW89] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A machine independent communication library. In J. Gustafson, editor, *Proc. Hypercube Concurrent Computers Conference*, 1989.

[Jel90] Robert Jellinghaus. Eiffel linda: An object-oriented linda dialect. *ACM SIGPLAN Notices*, 25(12):70–84, December 1990.

[Lel90] Wm Leler. Linda meets unix. *IEEE Computer*, pages 43–54, February 1990.

[Luc86] Steven E. Lucco. A heuristic linda kernel for hypercube multiprocessors. In *Proceedings of the Second Conference on Hypercube Multiprocessors*, pages 32–38, October 1986.

[MK88] Satoshi Matsuoka and Satou Kawai. Using tuple space communication in distributed object-oriented languages. In *Proceedings of OOPSLA '88*, pages 276–284. ACM, September 1988.

[Ste90] W. Richard Stevens. *Unix Network Programming.* Software Series. Prentice Hall, Englewood Cliffs, 1990.

[Sun90a] Sun Microsystems. *Network Programming Guide*, 1990.

[Sun90b] Sun Microsystems. *STREAMS programming*, 1990.

[WL88] Robert Whiteside and Jerrold Leichter. Using linda for supercomputing on a local area network. In *Proceedings Supercomputing '88*, pages 192–199, Orlando, Florida, November 1988. IEEE.