Aspectos da Implementação de Objetos Distribuídos*

Rogério Drummond† drummond@dcc.unicamp.br

Celso Gonçalves Júnior‡ celso@dcc.unicamp.br

Alexandre Prado Teles◊ teles@dcc.unicamp.br

Projeto A_HAND

Departamento de Ciência da Computação

Universidade Estadual de Campinas

Caixa Postal 6065, CEP 13081–970

Campinas, SP, Brasil

Dezembro de 1992

Apresentamos o trabalho desenvolvido pelo projeto A_HAND para a implementação de Objetos Distribuídos numa rede de computadores heterogêneos tendo UNIX como sistema operacional. Programação com objetos distribuídos é possível a partir de linguagens de programação adequadas e suporte para a criação e comunicação entre processos distribuídos. Discutiremos os conceitos fundamentais da área e as soluções adotadas pela nossa implementação.

We present the work developed at Projeto A_HAND to support distributed objects on a network of heterogeneous computers under UNIX operating systems. Distributed object programming is possible by means of proper programming languages and support to creation and communication among distributed processes. We discuss the fundamental concepts and the solutions we have adopted in our implementation.

---

* Esse trabalho é financiado pela Coordenadoria de Aperfeiçoamento de Pessoal de Ensino Superior (CAPES, processo n. 02022/92) e pelo Conselho Nacional de Pesquisa e Desenvolvimento (CNPq, processo n. 830131/92–2).

† PhD (*Cornell University*, 1986). Áreas de interesse: ambientes de desenvolvimento de *software*, sistemas distribuídos, *groupware*.

‡ BSc em Ciência da Computação (Unicamp, 1989); mestrando em Ciência da Computação (DCC – Unicamp). Áreas de interesse: arquitetura, sistemas distribuídos, linguagens de programação.

◊ BSc em Ciência da Computação (Unicamp, 1989); mestrando em Ciência da Computação (DCC – Unicamp). Áreas de interesse: compiladores, linguagens de programação, depuração, ambientes de desenvolvimento de *software*..

1. Introdução

A área de Sistemas Distribuídos tem recebido nos últimos anos considerável esforço de pesquisa, tanto pelas suas vantagens potenciais como pelos problemas apresentados. Pode-se dizer que essa área engloba todos os esforços empenhados em construir, a partir da disponibilidade crescente de recursos computacionais (processadores, memória e meios de comunicação), sistemas operacionais que gerenciam recursos de uma rede de máquinas autônomas interligadas.

Na área de *software* outro tema que tem recebido grande atenção é Programação Orientada a Objetos. Essa nova área apareceu como decorrência natural da constatação das debilidades das linguagens de programação tradicionais na construção de sistemas de *software* de porte e complexidade crescentes, num contexto de acelerada evolução do *hardware*.

Os conceitos de sistema distribuído e programação orientada a objetos têm vantagens e requisitos tais que tornam natural e desejável a integração entre os dois. A seguir daremos alguns fundamentos sobre esses assuntos.

2. Sistemas Distribuídos

O termo sistema distribuído é usado para designar um sistema que, executado simultaneamente em um conjunto de computadores — possivelmente heterogêneos —, permite a todos eles partilhar tarefas e recursos de maneira transparente, eficiente e segura. Esses computadores são autônomos, possuem memória privada e trocam informações através de um meio de transmissão disposto segundo alguma topologia. Ao "esconder" dos usuários que as tarefas e recursos estão divididos pela rede, o sistema se apresenta como um sistema centralizado, responsável pela gerência de todos os recursos da rede.

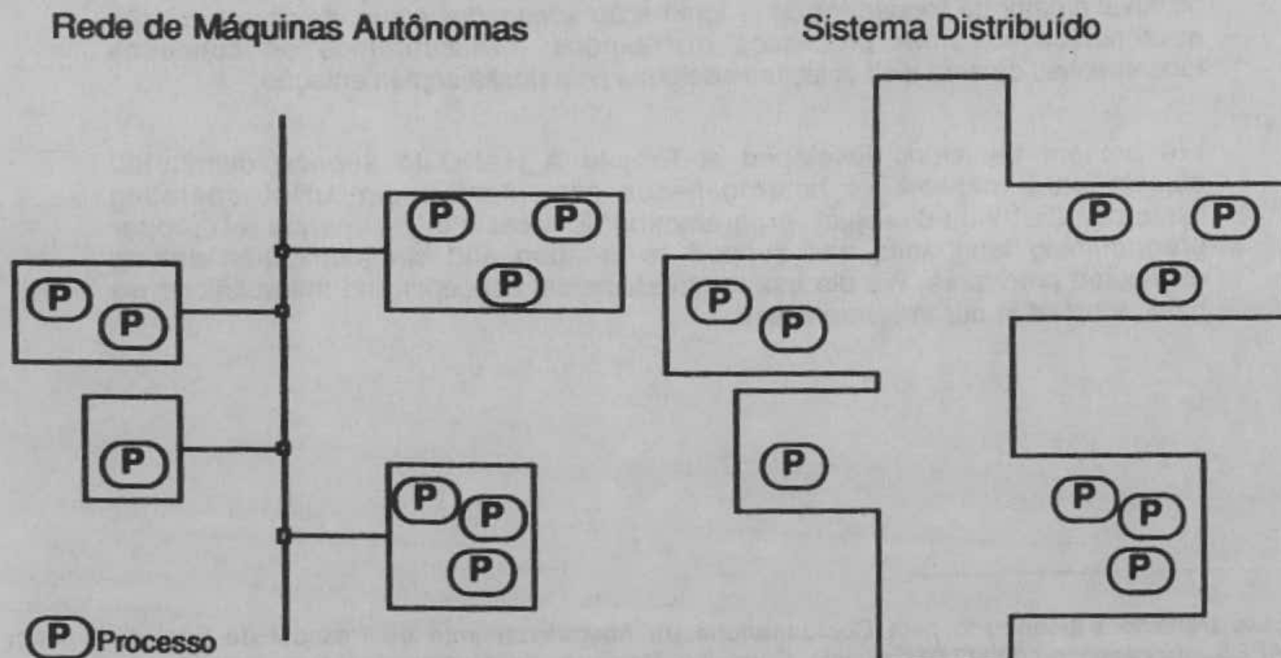

Figura 1

Tais sistemas se tornaram factíveis depois do aparecimento de redes de computadores ligados por um meio de comunicação de alto desempenho (taxa de transferência da ordem de 1MByte/s[1]). Esse modelo vem substituir o modelo de ambiente de programação tradicional, composto de um

[1]Medida típica de uma rede local, mas não há, *a priori*, restrição quanto ao tamanho da rede.

computador de grande capacidade de processamento e armazenamento (*mainframe*), que centraliza os recursos disponíveis (terminais, discos, impressoras, etc...). O novo modelo apresenta em relação ao modelo tradicional vantagens apreciáveis como: modularidade, extensibilidade e tolerância a falhas.

O conceito chave num sistema distribuído é transparência. Como regra mais geral, nesse sistema o usuário não sabe (ou não precisa saber) em qual máquina da rede estão armazenados seus arquivos ou estão sendo executados seus processos. Há muitos sistemas construídos para executar em uma rede de computadores, mas que não podem ser chamados de distribuídos porque não há transparência do ponto de vista das aplicações. O conceito de "sistema distribuído" está baseado mais em *software* do que em *hardware* [TRe85].

Vários sistemas distribuídos foram propostos e implementados; aqui podemos citar os sistemas Chorus [Arm89, Roz90], Mach [Acc86] e V [Che84, Che88]. Estes sistemas estão em uso já há vários anos, e estabeleceram uma base conceitual sólida e coerente para o desenvolvimento de trabalhos na área. Os sistemas Chorus e Mach, em particular, são de especial interesse porque o projeto de desenvolvimento de um sistema UNIX [RTh74] distribuído foi motivação para o trabalho de pesquisa que resultou nesses dois sistemas.

2.1 Sistema de arquivos

Numa rede de computadores os usuários geralmente desejam compartilhar as informações armazenadas nas máquinas da rede. Se o meio de comunicação é rápido o suficiente, as aplicações executadas em uma máquina podem usar informações mantidas em outra máquina rotineiramente e sem impacto sensível no desempenho. É importante destacar que nos referimos a um sistema onde um arquivo está guardado em um, e só um lugar; do contrário teríamos cópias espalhadas pela rede, com o consequente problema de consistência.

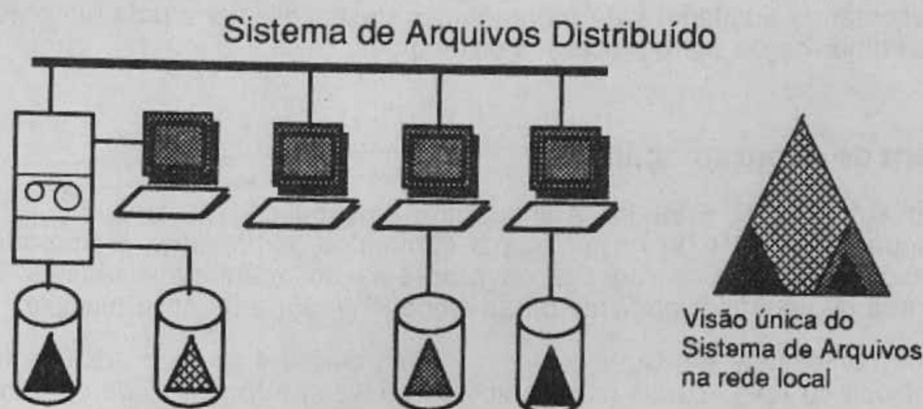

Figura 2

Por outro lado, tais acessos remotos não devem levar em conta a localização física do arquivo, nem ser diferentes de um acesso local (transparência). Isso pode ser feito por um sistema de arquivos que mapeia os sistemas de cada máquina num único sistema de arquivos, acessível desde qualquer ponto da rede — respeitadas as permissões de acesso. Tal sistema de arquivos deve desviar para a rede requisições de acesso a arquivos que não estão presentes na máquina do processo que fez a requisição, transmitir os dados pela rede de forma confiável e lidar com problemas como acesso simultâneo, queda de máquinas, etc...

2.2 Comunicação entre processos

Aplicações complexas podem ser decompostas em processos cooperantes para refletir melhor a estrutura dos programas, prover tolerância a falhas e, eventualmente, melhorar o desempenho — na possibilidade de explorar paralelismo. Tais processos trocam informações (estado do processo, informações produzidas externamente, mensagens de controle, etc...) e oferecem serviços uns aos outros.

Em um sistema distribuído, a localização dos processos (na memória de quais máquinas eles estão carregados) deveria, por regra geral, ser irrelevante para os mesmos; portanto, a troca de informações entre processos deve ser feita valendo-se exclusivamente dos recursos oferecidos pelo sistema operacional para esse fim. Isso pode introduzir um *overhead* na comunicação (processos colocados no mesmo espaço de endereçamento poderiam interagir através de memória compartilhada) mas traz várias vantagens: transfere responsabilidade do programador para o sistema operacional; simplifica as interfaces entre os processos cooperantes; e protege os dados privativos de cada processo contra acessos indevidos. Essa abordagem é coerente com o modelo de programação orientada a objetos [GRo83], já que o estado interno de um objeto só pode ser consultado/alterado via funções definidas na sua interface.

Os modelos mais usuais para comunicação entre processos são troca de mensagens [Hoa78] e chamada remota de procedimento[2] [Han78]. Esses modelos não serão detalhados aqui; mais adiante falaremos da sua utilização.

3. O Ambiente A_HAND

O nosso trabalho de implementação de Objetos Distribuídos faz parte do projeto A_HAND [DLi87] (Ambiente de desenvolvimento de *software* baseado em Hierarquias de Abstração em Níveis Diferenciados) do Departamento de Ciência da Computação da Unicamp. O objetivo maior do projeto é criar um ambiente voltado para o desenvolvimento de sistemas de *software* grandes e complexos. Para tal o ambiente deverá prover ferramentas que assistam a equipe de desenvolvimento durante todas as etapas da construção de um sistema.

Uma das características principais do projeto A_HAND é a de que objetos simples possam ser facilmente combinados na construção de objetos mais complexos, mantendo uma interface uniforme nos vários níveis de abstração.

Vamos apresentar as linguagens de programação desenvolvidas e uma biblioteca de funções para a criação e comunicação entre processos distribuídos.

3.1 Linguagem de programação Cm

A linguagem Cm [DSi88, Fur91] é a linguagem de produção do ambiente. É diretamente derivada da linguagem C [KRi78] no tocante a comandos, expressões e operadores. É uma linguagem definida segundo o paradigma de programação orientada a objetos, apresentando verificação rigorosa de tipos, polimorfismo paramétrico [CWe85] e herança múltipla.

A unidade de compilação em Cm é a classe. Uma classe é um tipo abstrato de dados, que encapsula definições de tipos, dados (constantes e variáveis) e funções que operam sobre esses dados. Uma classe pode ter parâmetros para ajustar sua funcionalidade ao ser usada em diferentes contextos. Um parâmetro de classe pode ser um valor (como um parâmetro comum de funções) ou um tipo (declarado como um parâmetro de tipo *type*). Parâmetros de tipo (*type*) podem ser usados como qualquer outro tipo (tanto pré-definido como definido na classe). Classes parametrizadas são ditas polimórficas, e definem, na realidade, infinitas classes (uma para cada possível combinação de valores para seus parâmetros).

Exemplo de classe polimórfica

```
Class Stack (type t, int n)
t [n] stack; int top = 0;

export void push (t value)
{
        if (top >= n)
                raise (StackOverflow);
        stack [top++] = value;
}
```

[2]Mais comumente referido como RPC (do inglês *Remote Procedure Call*).

```
export t Stack &pop ()
{
        if (top <= 0)
                raise (StackUnderflow);
        return (stack [--top]);
}
```

Figura 3

Objetos são instâncias de uma classe.

3.1.1 Portas

As operações de entrada e saída em Cm são realizadas através de portas de comunicação (ver 3.3). Essas portas são objetos da classe `Port`, que é pré-definida na linguagem. As portas são utilizadas para trocar mensagens entre os objetos que as possuem. Todas as portas de um objeto são visíveis externamente a ele, e compõe a sua interface com o mundo externo.

Declaração de uma classe

```
Class Merge ( ... )
      ...
      Port ( input )  in1, in2;
      Port ( output ) out;
      ...
```

Representação Gráfica

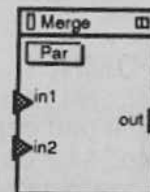

Figura 4

A representação gráfica é a usada pela LegoShell (ver 3.4). O botão colocado abaixo do nome do programa (rótulo `par`) permite a especificação dos parâmetros desse programa. As portas são representadas pelos triângulos colocados na borda do programa, rotulados pelo nome da variável correspondente.

3.1.2 Exceções

Qualquer anomalia na execução de um processo que impossibilite a continuidade ou exija um comportamento diferenciado de sua execução normal é chamada de exceção. Esta anomalia não é necessariamente um erro, mas indica um evento raro que deve ser tratado de forma particular. São exemplos de exceções: divisão por zero, referência a um apontador de valor ilegal, acesso a vetor com índice inválido, etc.

A linguagem Cm provê suporte para sinalização, propagação e tratamento de exceções. A implementação de exceções em Cm é bastante similar à da linguagem C++ [Str91]. Tratadores de exceções são associados estaticamente a blocos de comandos, porém são ativados dinamicamente, de acordo com o fluxo de execução do programa, podendo, inclusive, haver mais de uma ativação de um mesmo tratador (por exemplo, no caso de recursão). A propagação é automática em vários níveis, ou seja, se um dado tratador não pode tratar uma exceção, ela é automaticamente repassada ao tratador seguinte, e assim sucessivamente até ser encontrado um tratador ou não haver mais tratadores estabelecidos. Caso uma exceção não seja tratada, a execução do programa Cm é abortada.

Uma exceção pode ser propagada de um processo a outro através de portas de serviço (ver 4.3), caso o serviço requisitado na porta resulte em uma exceção no servidor.

Exceções em Cm podem "carregar" um valor de um dado tipo. Este tipo é usado para identificar a exceção e associá-la a tratadores. O valor é usado para passar informações relevantes do ponto de sinalização ao ponto de tratamento da exceção.

3.2 O Sistema de Arquivos

O sistema de arquivos utilizado em nosso sistema é o NFS [Sun90a], que permite a estruturação de um sistema de arquivos global, como o descrito anteriormente (ver 2.1).

Convém ressaltar que o NFS permite que cada máquina tenha sua própria visão do sistema de arquivos da rede, diferente das visões das demais máquinas. Este recurso, se bem administrado, é bastante vantajoso, estabelecendo um nível a mais de proteção de acesso (pode-se impedir que usuários de uma máquina acessem um dado sistema de arquivos, bastando para isso não incluir este sistema na visão local da máquina). Porém deve-se procurar evitar que um mesmo arquivo seja "visto" de maneira diferente por máquinas diferentes, isto é, deve-se procurar manter as diversas visões das diversas máquinas homogêneas e coerentes entre si (caso isso não seja feito, processos em máquinas diferentes não poderão trocar informações baseadas na visão local do sistema de arquivos, tais como *pathnames* de arquivos).

Com o servidor de nomes (ver 3.3) é possível tornar a localização de um arquivo (*pathname*, tanto com relação ao sistema de arquivos UNIX como com relação ao NFS) transparente ao usuário, bastando para isso cadastrar o arquivo e seu *vnode*[3] no servidor de nomes. Assim, um processo que deseje acessar o arquivo consulta o servidor para obter seu *vnode*.

3.3 O Sistema OMNI

As aplicações a serem desenvolvidas no nosso ambiente, assim como as ferramentas de auxílio ao desenvolvimento, são de natureza distribuída. Deve ser possível, portanto, a criação de processos espalhados pela rede e a troca de informações entre eles. O sistema UNIX oferece facilidades para tal, mas a programação a partir dessas ferramentas é penosa e não fornecem a transparência desejada, já que o UNIX não é um sistema distribuído no sentido pleno do termo.

O sistema OMNI [DCi92] de Suporte a Aplicações Distribuídas tem por objetivo facilitar a criação e comunicação entre processos dispersos numa rede de máquinas heterogêneas, cada máquina operando sob o sistema UNIX. O OMNI é constituído de um conjunto de programas que executam em cada máquina da rede (os *daemons* OMNI) e uma biblioteca de funções, que estendem conceitos do UNIX para o seu equivalente distribuído permitindo a construção de aplicações distribuídas com uniformidade, transparência e segurança.

O sistema OMNI está dividido em três módulos:

- Servidor de nomes: responsável pela identificação dos objetos dentro do sistema. A cada objeto é associada uma identificação — chamada `OMNIid` — única na rede e no tempo. Essa identificação será usada sempre que se queira efetuar uma operação sobre o objeto. A um objeto pode estar associado um nome simbólico, a ser usado como chave para a obtenção de sua `OMNIid`. O servidor oferece funções para cadastrar um nome, remover um nome e obter informações relativas a um nome. O servidor de nomes está distribuído por toda a rede oferecendo total transparência no acesso aos objetos do sistema.
- Gerenciador de processos: no sistema OMNI é possível criar processos segundo a hierarquia UNIX (pai – filho). Tais processos podem trocar sinais entre si, independentemente de sua localização. As funções oferecidas por esse módulo atuam num contexto distribuído e são funcionalmente equivalentes às *system calls* correspondentes. Por exemplo: as funções `om_forkExec` e `om_createProc` criam um

---

[3] *vnode* [Sun90a] é o descritor de arquivos usado pelo NFS, que basicamente contém uma referência ao sistema de arquivos UNIX onde se encontra o arquivo e seu *inode* (descritor tradicional do UNIX) neste sistema.

processo e retornam sua `OMNIid`, enquanto que `om_kill` envia um sinal usando como parâmetro a `OMNIid` do processo destinatário.

- Portas de comunicação: são o mecanismo utilizado para troca de informações entre processos cooperantes ou no esquema cliente – servidor. As portas podem ser de entrada, de saída, ou de entrada e saída simultaneamente, conforme a direção das mensagens que passam por ela. Um processo pode ter um número arbitrário de portas, dependendo de como é sua interação com outros processos. As portas podem exigir ou dispensar conexão, com o que serão denominadas conectáveis ou não-conectáveis, respectivamente. Uma porta conectável de saída deve ser conectada a uma porta conectável de entrada: essa conexão estabelece um canal por onde trafegarão as mensagens produzidas pelo processo dono da porta de saída e consumidas pelo processo dono da porta de entrada. Já as portas não-conectáveis só podem ser de entrada, e são tipicamente usadas para receber requisições de serviços. Uma requisição de serviço é enviada sem necessidade de passar por uma porta de saída. Caso o serviço pedido exija uma resposta, o processo que fez a requisição deve enviar também a referência de uma porta, para a qual será enviado o resultado da execução do serviço.

3.4 A linguagem de comandos *LegoShell*

A LegoShell [Dru89] é uma linguagem gráfica para a especificação de comandos complexos chamados computações. Uma de suas principais características é estender o conceito de *pipe* do UNIX. A LegoShell oferece conectores especiais com semântica de *broadcast* e *mailbox*, usados para concentrar e/ou disseminar informações produzidas pelos processos.

As computações são programas construídos a partir da conexão entre programas (tipicamente módulos em Cm), arquivos, dispositivos periféricos e conectores especiais. Podemos construir um exemplo de computação utilizando o programa `Merge` mostrado na figura 4. Esse programa faz um *merge* de suas duas entradas (portas `in1` e `in2`) e escreve o resultado na sua saída (porta `out`). A partir de um arquivo desordenado, podemos dividi-lo em duas partes, ordenar cada uma delas e fazer o *merge* das partes ordenadas. Para fazer a ordenação, vamos usar dois programas `Sort`; e para dividir o arquivo vamos usar um conector *mailbox*.

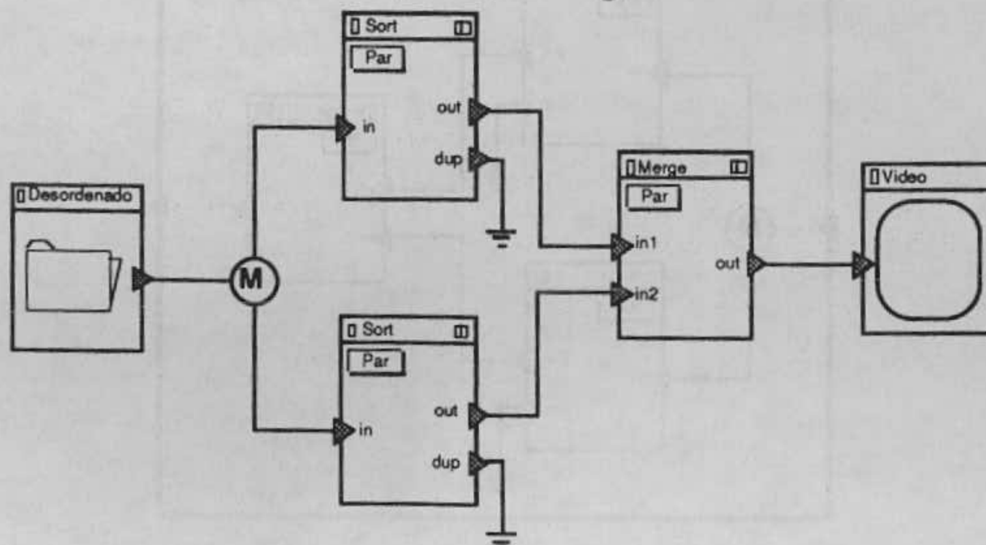

Figura 5

O programa `Sort` tem duas saídas, que produzem o mesmo resultado (`out` e `dup`). Se quisermos desprezar uma delas, basta conectá-la a um conector "buraco negro" — representado pelo sinal de "terra" —, que descarta todas as mensagens enviadas a ele. Esse conector é necessário para os casos onde é preciso "fechar" uma computação.

Cada um dos programas envolvidos pode receber parâmetros no instante de sua ativação (botão `par` abaixo do nome do programa). Deve-se notar que a interface de um programa com os demais objetos da computação resume-se às suas portas. Os arquivos terão portas compatíveis com suas permissões de acesso e com sua utilização dentro de uma computação; e cada dispositivo de entrada/saída utilizado também terá sua porta de entrada ou saída, conforme o caso. O conector *mailbox* recebe os dados de suas entradas e os distribui sob demanda (cada dado vai para apenas um consumidor; no conector *broadcast* vai para todos).

As computações podem ser programas distribuídos, o que significa que seus componentes podem estar localizados em diferentes máquinas da rede. Essa distribuição pode: derivar de algum aspecto ou requisito particular da aplicação; ser feita de forma a dividir a carga de processamento entre as máquinas; ou levar em conta a localização de algum recurso, a intensidade do fluxo de informação entre os processos, questões de segurança e tolerância a falhas, etc.

Um dos princípios fundamentais do projeto A_HAND é o de permitir que os objetos definidos no ambiente possam ser usados na definição de objetos mais complexos. Na LegoShell essa operação é chamada de abstração, e consiste em encapsular uma computação escondendo seus componentes e suas conexões; agora essa abstração pode ser usada como um programa em outras computações. A interface de uma abstração é composta de portas que não foram conectadas ou cuja conexão envolve componentes fora da abstração. Os parâmetros de uma abstração são referências para os parâmetros de seus componentes; se algum desses componentes for um programa Cm que requer um tipo como parâmetro (classe polimórfica), então essa abstração será uma computação polimórfica. A capacidade da LegoShell de preservar o polimorfismo da linguagem Cm vem do fato de ambas utilizarem a mesma estrutura de tipos; essa característica aumenta em muito a flexibilidade da linguagem e permite reutilização de código em larga escala.

Abstração da computação Mergesort

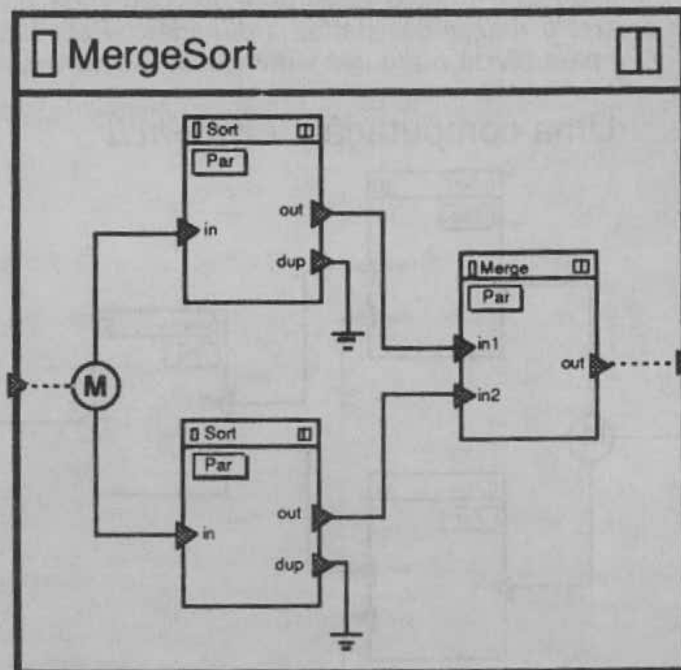

Figura 6

Aqui temos um novo programa chamado MergeSort, obtido a partir de uma abstração de parte da computação mostrada na figura 5. Esse programa possui uma porta de entrada e uma porta de saída, e faz ordenação dos dados de entrada.

Além da LegoShell temos uma outra linguagem de comandos, a CO^2 [Fer91]. Essa linguagem é textual e interpretada — semelhante às *shells* do UNIX —, com estruturas de controle para especificar comandos que estão fora do escopo da LegoShell. A CO^2 também têm estrutura de

tipos comum à linguagem Cm, sendo adequada para o desenvolvimento rápido de programas e construção de protótipos. Nessa linguagem é possível compor programas a partir da combinação de programas mais simples, de maneira análoga à construção de abstrações na LegoShell.

3.5 O Sistema de Execução (*Run Time System*)

O sistema OMNI dará suporte às linguagens de programação do A_HAND para a construção de aplicações distribuídas. As funções do sistema para a troca de mensagens têm semântica semelhante às operações de entrada/saída de baixo nível do sistema UNIX. É possível especificar ainda como os caracteres serão agrupados em mensagens: blocos de tamanho fixo ou variável, mensagens com caractere terminador, ou *streams*. Tais recursos são, todavia, inadequados para uso direto pelas linguagens de programações do ambiente, que enxergam as portas de comunição como objetos. O sistema OMNI será usado para dar suporte a aplicações muito diferentes entre si, portanto sua funcionalidade é bem geral; as abstrações previstas pelas linguagens de programação apresentadas, como portas tipadas[4] (ver 4.1), têm de ser implementadas acima do OMNI [Gon92].

Os mecanismos de comunicação entre processos devem ser oferecidos como recursos próprios das linguagens de programação preferencialmente à utilização de funções de biblioteca [BST89]. No nosso caso, o *Run Time System* da linguagem Cm terá por objetivo preencher o *gap* semântico entre o sistema OMNI e as linguagens de programação do ambiente. Essa abordagem visa aumentar o poder de expressão das linguagens, permitir a construção de aplicações robustas e unificar os procedimentos para uso dos recursos do OMNI.

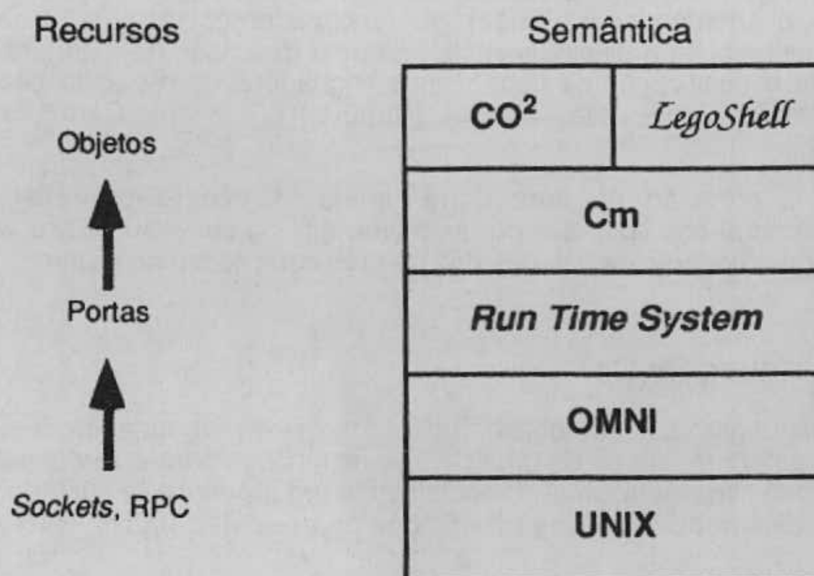

Figura 7

A linguagem Cm já se encontra em utilização dentro do projeto A_HAND, e atualmente está sendo estendida com diversos recursos novos, tais como sobreposição de operadores e funções[5], e tratamento de exceções [Tel92]. As extensões feitas a essa linguagem afetam as outras (LegoShell e CO^2) pois elas usam módulos escritos em Cm nos seus programas. Portanto as propriedades ou construções do Cm deveriam, sempre que possível, ser generalizadas para o

[4] Do inglês *typed ports*.

[5] Em inglês, *operator overloading* e *function overloading*.

resto do ambiente, visando uniformidade na construção das aplicações. O *Run Time System* deverá concentrar a funcionalidade que seja comum às linguagens e ajustar as interfaces entre elas.

O *Run Time System* também é responsável pelo gerenciamento de exceções, que engloba:

- ativação e desativação de tratadores;
- sinalização e propagação;
- representação de tipos em tempo de execução.

4. Aspectos de Implementação

Apresentamos agora alguns pontos sobre a implementação do *Run Time System* da linguagem Cm. Algumas questões ainda estão em aberto visto que o trabalho está em andamento.

4.1 Portas tipadas

Uma interessante especialização da classe `Port` é a classe `TypedPort`. As portas dessa classe têm um tipo associado e as operações de entrada/saída operam sobre objetos desse tipo: assim é possível aos processos trocar dados como registros, vetores, árvores, etc.

O uso dessas portas tipadas requer a transformação do dado numa sequência de *bytes* antes da transmissão e a transformação inversa na recepção; essas operações são denominadas linearização e deslinearização de estruturas, respectivamente. Essas operações não são triviais, pois usando apontadores pode-se criar estruturas recursivas (listas circulares ou árvores costuradas, p. ex.) cujo tratamento é complicado. No projeto A_HAND já existe uma ferramenta para esse trabalho, chamada Linear [Sou92]. As funções fornecidas pelo Linear necessitam, para as operações de linearização e deslinearização, de uma descrição da estrutura dos dados, escrita numa linguagem de especificação de tipos. Como linguagens de especificação de tipo, utilizadas para o mesmo fim, podemos citar a XDR [Sun90a] (*External Data Representation*) e a *Matchmaker* [JRa86].

Para realizar a conexão de uma porta tipada, é necessário fazer a verificação da compatibilidade estrutural dos tipos das portas envolvidas na conexão. Essa verificação pode ser facilitada se houver um controle de versões das classes compiladas no sistema.

4.2 Portas na Linguagem Cm

As portas da linguagem Cm são objetos utilizados pelos programas para se comunicar com o mundo externo. A estrutura interna de uma porta e as funções para a sua utilização estão escritas na classe `Port`. Toda classe em Cm é potencialmente um programa, e a interface de um programa são os parâmetros da sua classe e as variáveis do tipo `Port` declaradas. (Ver o programa `Merge` na figura 4.)

Antes de dar início à execução do programa, o *Run Time System* deverá preparar as portas para a sua efetiva utilização. As portas deverão ser criadas usando as funções do OMNI, sendo cadastradas no Servidor de Nomes quando necessário, e ao final do programa serão destruídas. A classe `Port` oferecerá as funções `open` e `close` para dar, respectivamente, início e término às operações numa porta. Essas operações, todavia, podem estar implícitas no começo e no fim do programa, da mesma maneira como ocorre com os arquivos de entrada padrão, saída padrão e saída padrão de erros dos processos UNIX.

O *Run Time System* provê uma porta de controle para cada processo. Essa porta é invisível para o usuário, sendo usada para receber e enviar mensagens de controle (`kill`, `ptrace`, requisições de serviços, exceções, p. ex.).

4.3 Portas de Serviço

Em uma rede existem programas que executam continuamente tarefas específicas, sob demanda dos usuários do sistema. Chamamos esses programas de servidores, e os programas que solicitam essas tarefas são os clientes. Os servidores centralizam o acesso a recursos da rede ou executam processamento complexo e muito especializado. Como exemplos de servidores podemos citar: servidor de X *Window*, gerenciador de banco de dados, servidores de disco e de impressão, etc.

Um servidor pode declarar uma porta através da qual receberá requisições do serviço que executa. Essa porta deverá ser do tipo não-conectável, pois não se sabe de antemão quais processos usarão o serviço; pode haver servidores cujo acesso é restrito por razões de segurança, mas em princípio qualquer processo é um cliente potencial. A essa porta também deverá estar associado um nome simbólico, nome pelo qual o serviço ficará conhecido na rede. Nesse contexto vamos chamar essas portas de "portas de serviço".

Na linguagem Cm, o pedido de um serviço é sintaticamente semelhante a uma chamada de procedimento. O nome do serviço é usado como nome do procedimento, e os parâmetros da chamada são os parâmetros da requisição. O nome do serviço usado na chamada pode ser diferente do nome pelo qual o serviço é conhecido; nesse caso é necessário fazer o *bind* dos dois nomes explicitamente no programa ou através da LegoShell.

O programa que implementa um servidor pode ser resumido a um *loop* infinito que executa basicamente duas operações: receber uma requisição de serviço; e executar esse serviço. Para receber uma mensagem enviada a uma porta não-conectável (tipo das portas de serviço) o OMNI fornece a primitiva `om_receive`. Essa função pode ser usada de forma a bloquear ou não o processo caso não haja mensagens disponíveis para leitura.

Uma outra forma de atender requisições é através do recebimento assíncrono de mensagens, o que abre a interessante possibilidade de exploração de concorrência dentro de objetos. Cada mensagem de requisição recebida na porta de serviço é mapeada em uma exceção, cujo tratador é um *dispatcher* que cria um processo específico para atender essa mensagem; esse processo terminaria sua execução após completar o serviço pedido. Cada um desses processos teria uma pequena área de dados local, e compartilharia com os demais o código e os recursos do servidor. "Processos leves"[6] [Sun90b, Pow91] serão utilizados para implementar essa funcionalidade.

Para explorar concorrência é preciso dotar a linguagem Cm de algum mecanismo de exclusão mútua, como monitores [Hoa74] ou semáforos [Dij68]. É importante salientar que esses mecanismos controlarão a concorrência dentro de objetos (os "processos leves" de um objeto estão no mesmo espaço de endereçamento); a sincronização entre objetos pode ser conseguida apenas através de trocas de mensagens. O mecanismo de "processos leves" a ser usado já suporta monitores, cabendo ao Cm prover suporte sintático, interface adequada e verificação do correto uso do recurso.

4.4 Tratamento de Exceções

Exceções são identificadas por seu tipo, logo deve haver um mecanismo de representação de tipos em tempo de execução. Esta representação deve considerar a classe em que a exceção foi definida, a estrutura do tipo em questão e a hierarquia de herança, caso o tipo seja uma classe. A solução adotada é gerar uma *string* de identificação para cada tipo, baseada exclusivamente nas características acima.

Na sinalização de uma exceção, sua "assinatura" é gerada e usada na propagação dela, comparando-se as identificações da exceção gerada com a aceita pelo tratador.

Se nenhum tratador foi encontrado, o *Run Time System* verifica se o processo foi ativado via uma porta de servico. Em caso afirmativo, a exceção é transferida pela porta (como "resultado" do serviço) e é sinalizada no processo que requisitou o serviço.

---

[6]Do inglês *Lightweight Processes*; também é usado o termo *thread*.

5. Conclusões

Apresentamos o trabalho sobre Objetos Distribuídos em andamento dentro do projeto A_HAND. Esse trabalho envolve as linguagens de programação Cm, LegoShell e CO^2, e o sistema OMNI de Suporte a Aplicações Distribuídas. Relativas a esse trabalho há quatro teses de mestrado em desenvolvimento dentro do Departamento de Ciência da Computação da Unicamp.

O Projeto A_HAND tem por objetivo criar um ambiente de desenvolvimento adequado para a construção de sistemas de *software* grandes e complexos. Nesse ambiente objetos complexos serão construídos a partir de objetos mais simples; mostramos como isso pode ser feito utilizando as linguagens de programação do ambiente. Além dessas linguagens, serão oferecidas ferramentas para que o desenvolvimento das aplicações se faça de maneira sistemática, distribuída e cooperativa.

Um ponto forte do ambiente é a integração entre as linguagens Cm, LegoShell e CO^2. Essas linguagens usam a mesma estrutura de tipos e o mesmo mecanismo de comunicação entre programas, de forma que é possível compor programas escritos em linguagens diferentes sem perder a capacidade de verificação da compatibilidade entre eles.

O Cm é a linguagem básica para construção de programas. Os programas escritos em Cm têm funcionalidade encapsulada e interface bem definida, estimulando a produção de código robusto, reutilizável e de alta qualidade. Através da LegoShell podemos especificar graficamente como os programas são combinados para criar programas mais complexos – as chamadas computações. A possibilidade de encapsular computações permite analisar como os componentes do sistema interagem nos seus diversos níveis de abstração. A CO^2 é usada para escrever programas que necessitam de estruturas de controle ausentes na LegoShell.

O nosso ambiente tem grande semelhança com os ambientes Conic [MKS89] e MP [MDu]. Também nesses ambientes a idéia básica para a construção de sistemas é: a construção de módulos, feita em linguagem de programação convencional (*"programming in the small"*), e as conexões entre esses módulos, escrita em uma linguagem de configuração (*"programming in the large"*). Essa abordagem [DKr76] ajuda a separar problemas específicos de implementação dos módulos da maneira como esses módulos serão interligados para construir um sistema completo.

Agradecimentos

Agradecemos a colaboração de Cassius Di Cianni, pela ajuda na revisão e contribuições para a versão final deste documento; e a Bill W. Coutinho, pelas mesmas razões, além das agradáveis sessões de aula do editor de texto.

Bibliografia

[Acc86] Accetta, M, Baron, R, Golub, D, Rashid, R, Tevanian, A. and Young, M. Mach: A new kernel foundation for UNIX development. Technical Report, Department of Computer Science, Carnegie Mellon University. Agosto de 1986.

[Arm89] Armand, F, Gien, M, Herrmann, F. e Rozier, M. Revolution 89 or "Distributing UNIX Brings it Back to its Original Virtues". In *Workshop on Experiences with Distributed (and Multiprocessor) Systems*, pp. 153–174. Ft. Lauderdale, FL, USA. Outubro de 1989.

[BST89] Bal, H. E, Steiner, J. G e Tanenbaum, A. S. Programming Languages for distributed operating systems. *ACM Computing Surveys 21*, 3: pp. 261–322. Setembro de 1989.

[Che84] Cheriton, D. The V Kernel: A software base for distributed systems. IEEE Software. Abril de 1984, pp 19-42.

[Che88] Cheriton, D. The V distributed system. *Communications of the ACM 31*, 3: pp. 314–333. Março de 1988.

[CWe85] Cardelli, L. e Wegner, P. On Understanding types, Data Abstraction, and Polymorphism. *ACM Computing Surveys 17*, 4: pp. 471–522. Dezembro de 1985.

[DCi91] Drummond, R. e Di Cianni, C. Sistema de Execução para a linguagem Legoshell. DCC, Unicamp. Abril de 1991.

[Dij68] Dijkstra, E. W. Cooperating Sequential Processes. In *Programming Languages*. Academic Press, New York, NY. 1972.

[DKr76] DeRemer, F. e Kron, H. H. Programming–in–the–Large Versus Programming–in–the–Small. *IEEE Transactions on Software Engineering SE-2*, 2: pp. 80–86. Junho de 1976.

[DLi87] Drummond, R. e Liesenberg, H. A_HAND Ambiente de desenvolvimento de software baseado em Hierarquias de Abstração em Níveis Diferenciados. Em *IV Encontro de Trabalho do Projeto Ethos*, Petrópolis, RJ. Abril de 1987.

[Dru89] Drummond, R. Legoshell Linguagem de computações. Em *III Simpósio Brasileiro de Engenharia de Software*, Recife, PE. Outubro de 1989.

[DSi88] Drummond, R. e da Silva, F. B. Q. Linguagem Cm: Manual de Referência. Anais da IV Reunião de Trabalho do Projeto ESTRA, pp. 175–210. SID Informática. Outubro de 1988.

[Fer91] Fernandez, M. Linguagem de Comandos para Desenvolvimento de Software. Proposta de Tese de Mestrado, DCC, Unicamp. Outubro de 1991.

[Fur91] Furuti, C. A. Um compilador para uma linguagem de programação orientada a objetos. Tese de Mestrado, DCC, Unicamp. Julho de 1991.

[Gon92] Gonçalves Jr, C. Suporte para Programação em Sistemas Distribuídos. Proposta de Tese de Mestrado, DCC, Unicamp. Março de 1992.

[GRo83] Goldberg, A. e Robson, D. *Smalltalk–80: The Language and its Implementation.* Addison–Wesley, Reading, MA, USA. 1983.

[Han78] Hansen, P. B. Distributed Processes: A Concurrent Programming Concept. *Communications of the ACM 21*, 11: pp. 934–941. Novembro de 1978.

[Hoa74] Hoare, C. A. R. Monitors: An Operating System Structuring Concept. *Communications of the ACM 17*, 10: pp. 549–557. Outubro de 1974.

[Hoa78] Hoare, C. A. R. Communicating Sequential Processes. *Communications of the ACM 21*, 8: pp. 666–677. Agosto de 1978.

[JRa86] Jones, M. B. e Rashid, R. F. Mach and Matchmaker: Kernel language support for object oriented distributed systems. *OOPSLA `86 Conference Proceedings, Portland, OR, USA*. Setembro de 1986.

KRi78] Kernighan, B. W. e Ritchie, D. M. *The C Programming Language.* Prentice–Hall, Inc. 1978.

[MDu] Magee, J. e Dulay, N. MP: A Programming Environment for Multicomputers. Submited for publication.

[MKS89] Magee, J, Kramer, J. e Sloman, M. Constructing Distributed Systems in Conic. *IEEE Transactions on Software Engineering 15*, 6: pp. 663–675. Junho de 1989.

[Pow91] Powel, M. L, Kleiman, S. R, Barton, S, Shah, D, Stein, D. e Weeks, M. SunOS multi-thread architecture. Technical Report, Sun Microsystems Inc. 1991.

[Roz90] Rozier, M, Abrossimov, V, Armand, F, Boule, I, Gien, M, Guillemont, M, Herrmann, F, Kaiser, C, Langlois, S, Léonard, P e Neuhauser, W. Overview of the Chorus Distributed Operating System. Technical Report CS/TR–90–25, Chorus Systèmes. Abril de 1990.

[RTh74] Ritchie, D. M. e Thompson, K. The UNIX time-sharing system. *Communications of the ACM* 17 (7), pp 365-375. Julho de 1974.

[Sou92] de Souza, M. A. H. Relatório Parcial de Projeto de Bolsa de Iniciação Tecnológica e Industrial: Sistema Linear. DCC, Unicamp. Agosto de 1992.

[Str91] Stroustrup, B. *The C++ Programming Language (Second Edition)*. Addison–Wesley, Reading, MA, USA. 1991.

[Sun90a] Sun Microsystems. Network Programming Guide, Sun Microsystems Inc. Março de 1990.

[Sun90b] Sun Microsystems. Programming Utilities & Libraries, Sun Microsystems Inc. Março de 1990.

[Tel92] Teles, A. P. Extensões à Linguagem Cm. Proposta de Tese de Mestrado DCC, Unicamp. Março de 1992.

[TRe85] Tanenbaum, A. S. e Renesse, R. V. Distributed Operating Systems. *ACM Computing Surveys 17*, 4: pp. 419–469. Dezembro de 1985.