OMNI
Sistema de Suporte a Aplicações Distribuídas

Cassius Di Cianni

DCC – IMECC – UNICAMP

Departamento de Ciência da Computação
Instituto de Matemática, Estatística e Ciência da Computação
Universidade Estadual de Campinas

OMNI
Sistema de Suporte a Aplicações Distribuídas

Cassius Di Cianni

Este exemplar corresponde à redação final da tese corrigida e defendida pelo Sr. Cassius Di Cianni e aprovada pela Comissão Julgadora.

Cidade Universitária Zeferino Vaz, Campinas, 23 de setembro de 1994

Prof. Dr. Rogério Drummond Burnier Pessôa de Mello Filho

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação, da Universidade Estadual de Campinas, como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Cianni, Cassius Di

C481o OMNI sistema de suporte a aplicações distribuídas / Cassius Di Cianni -- Campinas, [S.P. :s.n.], 1994.

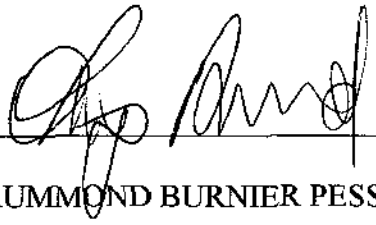
Orientador : Rogerio Drummond Burnier Pessoa de Mello Filho

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Matemática, Estatística e Ciência da Computação.

1. Sistemas operacionais distribuídos (Computadores). 2. Processamento eletrônico de dados - Processamento distribuído. 3. Sistemas abertos (Computadores). I. Mello Filho, Rogerio Drummond B.P. de (Rogerio D.B.P.de)II. Universidade Estadual de Campinas. Instituto de Matemática, Estatística e Ciência da Computação. III. Título.

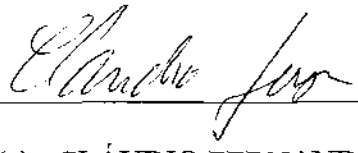
Tese de Mestrado defendida e aprovada em 23 de Setembro de 1994

pela Banca Examinadora composta pelos Profs. Drs.



Prof (a). Dr (a). ROGÉRIO DRUMMOND BURNIER PESSOA DE MELLO FILHO

Prof (a). Dr (a). RICARDO DE OLIVEIRA ANIDO



Prof (a). Dr (a). CLÁUDIO FERNANDO RESIN GEYER

A Solange e a meus pais,
pela força...

Agradecimentos

O autor gostaria de agradecer às seguintes pessoas ou instituições, sem cujo apoio este trabalho não poderia ter sido realizado:

Solange, minha noiva, pela força, carinho, compreensão e paciência,

Toda a minha família, em especial meus pais e irmãs, pela força, carinho, compreensão, paciência e grana,

Meus companheiros do Projeto A_HAND, que contribuíram com idéias, críticas e sugestões sem as quais o sistema jamais teria chegado até aqui. Em especial gostaria de agradecer a Bill Coutinho, Celso Gonçalves Jr., Alexandre Teles, Carlos Furuti, Mauricio Fernández e, é claro, ao maior "culpado" de todos, meu orientador Rogério Drummond,

Alunos das turmas de MC 505 e MC 705 do 2° semestre de 1991 ao 1° de 1994, que sentiram o OMNI na pele,

Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), que financiou parte do projeto através de uma bolsa de mestrado de dois anos, processo n° 91/1823-0.

"The computer IS the network, the network IS the computer."

H. A. Livingstone, Cyberspace Chronicles, 2004.

Resumo

O Sistema OMNI oferece facilidades para a criação, comunicação e gerenciamento de processos numa rede heterogênea de computadores com o sistema operacional UNIX. Ele estende os conceitos presentes no UNIX oferecendo serviços similares para um sistema distribuído. Seus três principais módulos são descritos: *Servidor de Nomes*, *Módulo de Portas* e *Gerenciador de Processos*.

Cada entidade do sistema, tais como portas, processos e grupos de portas, tem uma identificação única e homogênea, chamada de OMNIid, capaz de identificá-lo no tempo e no espaço. O Servidor de Nomes é responsável por associar nomes definidos pelo usuário a essas identificações, de tal forma que elas possam ser mais tarde recuperadas por outros processos em qualquer ponto da rede, provendo transparência quanto à real localização das entidades.

O Módulo de Portas de Comunicação provê o mecanismo usado por processos para se comunicarem através do envio de mensagens a portas. Portas podem ser com ou sem conexão. Uma mensagem só pode ser enviada a uma porta conectável pela porta a ela conectada, ao passo que qualquer processo pode enviar uma mensagem a uma porta sem conexão. O sistema também suporta *conectores especiais* e *grupos de portas*, que permitem que uma mensagem seja enviada simultaneamente a muitas portas ou seja entregue a apenas uma de um conjunto de portas.

O Gerenciador de Processos é responsável por criar processos distribuídos, enviar sinais a eles, perceber e relatar seu término. A maioria dos serviços do UNIX são estendidos, permitindo que processos sejam interrompidos por um sinal quando um processo filho morre ou que bloqueiem até seu término. Grupos de processos e sessões do UNIX são também estendidos.

Abstract

The OMNI system provides facilities for the creation, communication and management of processes in an heterogeneous network of computers running UNIX. It extends UNIX concepts by providing similar services on a distributed system. The system's three main modules, called the *Name Server*, the *Communication Ports Module* and the *Process Manager* are briefly described.

Every system entity, such as ports, processes or port groups, has a unique, homogeneous, system-wide identification called OMNIid, which identifies it in both time and space. The Name Server is responsible for associating user-defined names with these identifications, so that they can be later retrieved by other processes anywhere on the network, thus providing transparency with regard to actual entity location.

The Communication Ports Module provides a mechanism used by processes to communicate with one another by sending messages to ports. Ports can be connection or connectionless. A message can only be sent to a connection port by its connected peer, while any process can send a message to a connectionless port. The system also supports *special connectors* and *port groups*, which allow a message to be multicast to many ports or to be delivered to one of several possible ports only.

The Process Manager is responsible for creating distributed processes, sending signals to them, detecting and reporting their termination. Most UNIX services are extended, allowing parent processes to be interrupted by a signal at child's death or block until child termination. UNIX process groups and sessions are also extended.

Índice

Capítulo 1

Introdução

O Sistema OMNI [DC92] encontra-se em desenvolvimento no âmbito do Projeto A_Hand [DL87a, DL87b], do Departamento de Ciência da Computação (DCC) do Instituto de Matemática, Estatística e Ciência da Computação (IMECC) da Universidade Estadual de Campinas (UNICAMP). Seu objetivo é oferecer suporte ao desenvolvimento de aplicações distribuídas numa rede heterogênea de computadores com sistema operacional UNIX. Com ele é possível criar, gerenciar e comunicar processos distribuídos de maneira uniforme e transparente através da rede. O Sistema OMNI procura estender os conceitos presentes no UNIX de forma a dar a ilusão ao usuário de que ele se encontra numa plataforma com sistema operacional realmente disribuído. Contudo, nenhuma alteração foi feita ao núcleo do UNIX, a fim de manter a portabilidade do sistema.

Neste capítulo é discutida a motivação por trás da criação do OMNI, quais os requisitos desejáveis para um ambiente de programação distribuída e como o sistema se relaciona às demais camadas do ambiente A_Hand. Uma visão geral do sistema é apresentada no capítulo 2.

1.1 Motivação

A motivação para o desenvolvimento do Sistema OMNI deve ser entendida dentro do contexto do Projeto A_Hand. Este compõe-se de cerca de trinta pessoas, entre professores doutores, mestres, mestrandos, bacharéis ou engenheiros em computação e alunos de graduação. A meta desta equipe é pesquisar e desenvolver novas tecnologias a fim de produzir *software* de ponta com qualidade internacional.

Uma das principais áreas de atuação do Projeto A_Hand é a de sistemas distribuídos. Entretanto, as ferramentas disponíveis no UNIX para o desenvolvimento de tais sistemas não provêem o grau de abstração necessário para permitir que aplica-

ções distribuídas sejam implementadas rápida e confiavelmente. As ferramentas existentes são:

- Facilidades de Transporte: mecanismos como *sockets* [NPG90] ou TLI [TLI93] (*Transport Level Interface*) permitem atribuir um ou mais endereços de comunicação a processos UNIX. Se o processo *A* conhece o endereço de *B*, ele pode tentar estabelecer uma conexão com *B*, que deverá por sua vez aceitar o pedido de *A*. Uma vez conectados, os processos podem trocar informações à vontade. Os processos podem também comunicar-se através de datagramas, sem estabelecimento de conexão e sem garantia de entrega, duplicação ou ordem relativa das mensagens;
- RPC (*Remote Procedure Call*) [NPG90]: implementado sobre as facilidades de transporte, o mecanismo de Chamada Remota de Procedimento permite que um processo cliente invoque um procedimento de um processo servidor como se este fosse local. Antes de mais nada, entretanto, o cliente deve informar à biblioteca de RPC a localização, o número de programa e a versão do servidor.

Estas facilidades não são suficientes para o desenvolvimento eficiente de aplicações distribuídas pois o nível de abstração que oferecem é baixo demais. Para tanto seria desejável que o ambiente provesse os seguintes recursos:

1. Heterogeneidade e Interoperabilidade: idealmente, o ambiente deve ser totalmente independente de arquitetura e sistema operacional. Em nosso caso particular, nos contentaremos em idealizar um ambiente que opere sobre qualquer UNIX padrão, não importando o vendedor ou o *hardware*. Objetos devem interagir entre si de forma independente da plataforma onde se encontram;
2. Portabilidade: aplicações escritas em tal ambiente devem ser facilmente portáveis para outras plataformas onde o ambiente está implementado. O ambiente em si também deve ser facilmente portável;
3. Facilidade de uso: o ambiente deve, na medida do possível, livrar o programador de ter que se preocupar com os aspectos da distribuição de seu programa, permitindo que ele se concentre no desenvolvimento da aplicação propriamente dita;
4. Orientação a objetos: o paradigma de orientação a objetos nos permite separar claramente a funcionalidade e a interface de um componente de *software* de sua implementação, algo imprescindível num sistema heterogêneo. Outras vantagens não relacionadas diretamente ao fato do sistema ser distribuído também contribuem para tornar o paradigma atraente, como maior clareza do código e maior facilidade de reutilização do *software*;
5. Transparência quanto à localização: facilidades como RPC, *sockets* ou TLI exigem que um processo explicitamente especifique a localização do processo com o qual deseja comunicar-se. O ideal é que o ambiente de alguma maneira "encontre" no sistema o objeto desejado e se encarregue de efetuar a comunicação;
6. Agrupamento de objetos: muitas vezes é útil possibilitar que um objeto se comunique simultaneamente com todos os objetos dentro de um mesmo grupo ou com um desses objetos, escolhido (de preferência pelo sistema) dentro do grupo com base em algum critério;

7. Segurança: o sistema deve prover mecanismos que mantenham sua integridade e privacidade impedindo que usuários venham a adquirir privilégios que não possuem;
8. Tolerância a falhas: falhas são um fato do dia-a-dia. O sistema deve ser robusto o suficiente para permitir que aplicações nele escritas disponham de uma maneira adequada de detectar e se recuperar de falhas;

Estes são os requisitos ideais de um ambiente de programação distribuída. Em maior ou menor grau, todos eles deverão estar presentes na versão final do Ambiente A_Hand. Obviamente não é viável que uma única camada de *software* ofereça toda essa funcionalidade. O Sistema OMNI é apenas a primeira camada acima do UNIX. Vejamos como o OMNI e os demais níveis se integram para formar o núcleo do ambiente A_Hand.

1.2 O Papel do OMNI

O principal objetivo do OMNI é criar a ilusão de que o UNIX é um sistema operacional distribuído, mas que no mais continua sendo, essencialmente, o UNIX ao qual o programador está acostumado. Os conceitos do UNIX que foram estendidos são:

- Processos: no UNIX um processo pode criar outro na mesma máquina. No OMNI ele pode criar outro em qualquer máquina. O OMNI entende o processo criador como sendo pai do processo criado, e o pai pode esperar a morte de seu filho, da mesma maneira que no UNIX. O conceito de grupos de processos também é suportado;
- Sinais: é possível enviar sinais a processos distribuídos de maneira equivalente à forma que o UNIX o faz a processos locais, inclusive a grupos de processos. Quando um processo morre, seu pai pode receber um sinal informando-o de seu término;
- *Pipes*: este conceito foi estendido para o caso mais geral de Portas de Comunicação. Tais portas oferecem (muito) mais funcionalidade do que *pipes*, mas um *pipe* distribuído poderia ser considerado um caso particular delas. Por exemplo: no UNIX a *shell* (na verdade, qualquer programa) é capaz de redirecionar a saída padrão de um processo para a entrada padrão de outro sem que os processos envolvidos tomem conhecimento disso. No OMNI é possível fazer o mesmo com portas de comunicação, isto é, um terceiro processo (equivalente à *shell* do nosso exemplo) é capaz de conectar as portas de dois outros. É possível também especificar que um descritor de um processo (por exemplo a entrada ou a saída padrão) seja compreendido como uma porta, o que torna possível comunicar processos distribuídos mesmo que eles não tenham conhecimento do OMNI;
- Sistema de arquivos: o OMNI não entra nessa questão, pois considera que o NFS (*Network File System*) [NPG90] já provê este serviço.

O OMNI introduz também alguns outros conceitos não presentes no UNIX. Entre eles destacam-se algumas das facilidades oferecidas pelas Portas de Comunicação e pelo Servidor de Nomes. Por enquanto diremos apenas que as portas de comunicação oferecem uma maneira de permitir a comunicação entre dois ou mais processos, e que o Ser-

vidor de Nomes possibilita associar uma estrutura de dados a um nome simbólico, permitindo que essa informação seja recuperada de qualquer ponto da rede. Discutiremos maiores detalhes oportunamente.

1.3 Algumas definições

Definimos aqui alguns termos que utilizaremos mais adiante:

- Módulo: arquivo texto contendo o código fonte de uma parte (ou mesmo de todo) um programa. Dependendo do contexto, podemos também nos referir a um módulo como sendo uma parte relativamente auto-contida de um sistema de *software*, como por exemplo, os módulos do Sistema OMNI (Servidor de Nomes, Portas de Comunicação e Gerenciador de Processos);
- Módulo objeto: resultado da compilação de um módulo de um programa;
- Programa: arquivo executável resultante da ligação (*linking*) de todos os módulos objeto que compõem o programa;
- Processo: programa em execução;
- Programa distribuído: conjunto de programas que, quando em execução, comunicam-se entre si a fim de efetuar alguma tarefa;
- Programa distribuído em execução: conjunto de processos executando em máquinas (potencialmente) diferentes e cooperando entre si para a realização de uma tarefa;
- Entidade OMNI: qualquer abstração provida pelo Sistema OMNI que possa ser referenciada por sua identificação OMNI (OMNIid), tais como portas de comunicação, processos, grupos de portas ou grupos de processos;
- Objeto: encapsula uma estrutura de dados e oferece operações que podem ser efetuadas sobre tais dados, chamadas métodos. De fora de um objeto, a única maneira de fazer acesso a seus dados ou requisitar a execução de algum serviço é através da ativação de um método. Um objeto pode ser implementado por um módulo, um programa, ou um programa distribuído;
- Objeto remoto: objeto executando (ou que irá ser executado) em um processo diferente do processo do objeto que o utiliza. A menos de ter que declará-lo explicitamente como sendo remoto, um dado objeto utiliza um objeto remoto exatamente da mesma forma que utiliza um local.

1.4 Níveis acima do OMNI

O OMNI foi idealizado principalmente para dar suporte a três outras linguagens de programação em desenvolvimento no projeto A_Hand:

- Cm [Fr91, Tl93, Gn94], uma extensão da linguagem C orientada a objetos, com verificação forte de tipos, controle de exceções e amplo suporte à programação modular. A linguagem usará o OMNI para incorporar também suporte a programação distribuída;

- *LegoShell* [Dr89], uma linguagem gráfica de configuração de programas distribuídos. que permite interligar as portas de comunicação de programas Cm ou mesmo de programas executáveis quaisquer (os descritores de arquivo padrão de um programa qualquer podem ser encarados como portas de comunicação). Permite também resolver referências a objetos remotos;
- CO^2 [FD91, Fd94], uma linguagem de comandos (*shell*) voltada à prototipagem rápida de programas distribuídos. CO^2 engloba toda a funcionalidade da *LegoShell* em uma linguagem textual, fornecendo ainda muitos outros recursos que não podem ser satisfatoriamente oferecidos por uma linguagem gráfica, como comandos condicionais ou de repetição, por exemplo.

Uma vez que o ambiente de programação do A_Hand esteja implementado, raramente será necessário que o programador se utilize do OMNI diretamente. Idealmente, ele escreverá cada um de seus módulos em Cm, que lhe oferece facilidades de programação modular, orientação a objetos, verificação forte de tipos e controle de exceções. Os objetos executando em cada processo comunicam-se com objetos em outros processos através dos mecanismos de ativação remota de métodos ou de portas tipadas de comunicação.

A conexão entre as portas de comunicação dos vários processos (e a concomitante verificação de tipos) e mesmo a resolução das referências feitas a objetos remotos dentro de um objeto local podem ser feitas pela *LegoShell* ou CO^2, dando liberdade ao desenvolvedor para programar em dois níveis:

- Nível de *programação*, onde é utilizado Cm (preferencialmente) para escrever os objetos que compõem os módulos e os programas;
- Nível de *configuração*, onde é utilizado *LegoShell* ou CO^2 para interligar as portas de comunicação e resolver as referências indefinidas a objetos remotos;

Essa divisão em dois níveis propicia um maior grau de liberdade e facilita a reutilização do mesmo programa em diversos programas distribuídos.

As camadas do ambiente de desenvolvimento A_Hand estão mostradas na figura 1.

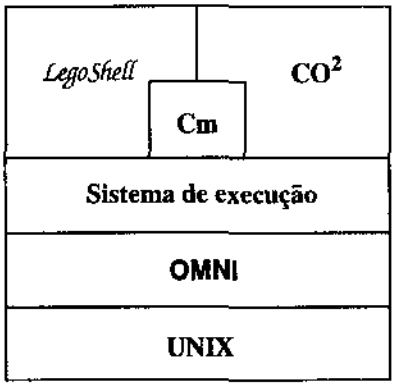
Cada uma das três linguagens possui seu sistema de execução (*run time system*) próprio. O "Sistema de execução" mostrado no diagrama é uma fatoração das partes comuns aos sistemas das três linguagens, e é através dele que é feita a interface entre as linguagens e o OMNI.

1.5 Organização deste Trabalho

No restante deste trabalho continuamos a apresentar o OMNI da seguinte forma:

- O capítulo 2 oferece uma visão geral do sistema, expondo a sua divisão em módulos e mostrando quais os principais serviços fornecidos por cada módulo;
- No capítulo 3 são apresentadas outras facilidades existentes na área de sistemas distribuídos, sendo traçadas comparações com o OMNI;

FIGURA 1 Diagrama simplificado das camadas do ambiente de programação A_Hand.



- O capítulo 4 descreve o módulo Servidor de Nomes, responsável por associar nomes simbólicos às identificações OMNI, que descrevem as entidades do sistema;
- No capítulo 5 são descritas as Portas de Comunicação, que são o mecanismo de troca de mensagens oferecido pelo OMNI;
- O capítulo 6 apresenta o módulo Gerenciador de Processos, responsável pela criação e troca de sinais entre processos distribuídos;
- No capítulo 7 é relatada a experiência resultante da implementação de um protótipo do sistema;
- Por fim, no capítulo 8, são traçadas as conclusões obtidas a partir deste trabalho.

Capítulo 2

Visão Geral do Sistema OMNI

O Sistema OMNI compõe-se de três módulos principais: o *Servidor de Nomes*, o *Gerenciador de Processos* e o *Módulo de Portas*, sendo que este último é subdividido em *Portas Conectáveis* e *Não-Conectáveis*. Cada módulo é responsável por prover uma parte da funcionalidade do sistema, tanto ao usuário quanto a suas outras camadas.

O OMNI é implementado por uma biblioteca de funções escritas em C [KR78, KR88] e por um conjunto de *daemons* que executam em cada máquina da rede.

Cada entidade do sistema (como processos, portas de comunicação, etc) pode ser referenciada através de sua identificação OMNI, ou OMNIid. Essa estrutura identifica a entidade unicamente no tempo e no espaço, ou seja, nunca serão atribuídas duas OMNIids iguais a duas entidades distintas, mesmo que elas não coexistam na rede. De posse dessa identificação, o usuário pode manipular a entidade por ela representada não importando sua localização. Maiores detalhes sobre a OMNIid são dados na seção 4.1.

2.1 O Servidor de Nomes

A camada mais baixa do sistema é o Servidor de Nomes. Ele não utiliza nenhum dos serviços das camadas superiores mas é por elas utilizado. Normalmente o usuário do OMNI não faz uso do Servidor de Nomes diretamente, usando-o apenas de forma indireta através das camadas de Portas de Comunicação e Gerenciamento de Processos. Contudo, tem a possibilidade de fazê-lo, se desejar.

A principal função do Servidor de Nomes é oferecer transparência quanto à localização das entidades OMNI. Isso é obtido associando-se um some simbólico (seqüência de caracteres) às OMNIids das entidades. O par (nome, OMNIid) fica registrado no Servidor e a OMNIid pode ser recuperada uma vez fornecido o nome,

pouco importando a localização da entidade ou a do processo que fez a consulta ao Servidor.

Os principais serviços providos pelo Servidor de Nomes são:

- Registrar um par (nome, OMNIid);
- Recuperar a OMNIid uma vez fornecido o nome;
- Remover um par (nome, OMNIid).

O Servidor permite também ajustar os valores de vários atributos dos nomes, como por exemplo:

- visibilidade: diz se o nome pode ser consultado fora da máquina local;
- unicidade: diz se um nome é único na rede ou se o mesmo nome pode ser associado a diferentes OMNIids;
- permissões de acesso: diz quem pode recuperar a OMNIid a partir do nome.

2.2 Portas de Comunicação

Portas são o mecanismo de comunicação do OMNI que permite a troca de mensagens entre os processos e são descritas em detalhe no capítulo 5. Uma mensagem constitui-se de uma seqüência de *bytes* de tamanho finito (porém arbitrário). Dois tipos de portas são suportados: *Portas Conectáveis* e *Não Conectáveis*.

2.2.1 Portas Conectáveis

Portas conectáveis estendem o conceito de *pipes* presente no UNIX. Com elas é possível criar conexões uni ou bidirecionais entre dois processos ou entre um processo e um conector especial (explicado mais adiante). Dessa forma, o que um processo escreve em uma ponta da conexão é recebido na outra.

No UNIX é possível para um processo criar um *pipe* interligando dois outros, como, por exemplo, quando a *shell* executa o comando "`ls | more`". Note que a conexão entre os dois processos (isto é, o *pipe*) foi estabelecida pela *shell* e não pelos processos. No OMNI, o mesmo pode ser feito com portas conectáveis, i. e., um processo pode interligar as portas de dois outros.

2.2.1.1 Conectores Especiais

Conectores especiais são uma maneira de permitir que muitas portas sejam conectadas a muitas outras, ao invés de apenas uma ser conectada a outra. Podemos, por exemplo, conectar três portas de saída à entrada de um conector e a saída do conector a cinco portas de entrada. Isso faz com que os dados produzidos pelas três portas de saída possam ser recebidos nas cinco de entrada. Três tipos de conectores são suportados:

- *Broadcast*: dissemina tudo que recebe em sua entrada a todas as portas ligadas a sua saída,

- *Mailbox*: uma mensagem recebida na entrada é repassada somente a uma as portas conectadas à saída,
- Conector local: é um conector de *broadcast* ou *mailbox* utilizado para disseminar dados para os (ou provenientes dos) descritores de arquivo do processo que o criou.

2.2.2 Portas Não Conectáveis

Esta facilidade permite enviar mensagens entre processos sem que seja necessário estabelecer uma conexão. Estas portas não garantem que uma mensagem seja entregue, ou que apenas uma cópia da mensagem seja recebida. Também não é possível afirmar nada sobre a ordem em que as mensagens chegarão. Entretanto, o uso destas portas pode ser mais eficiente do que o das Portas Conectáveis para certas aplicações.

2.2.2.1 Grupos de Portas

Assim como existem conectores especiais para Portas Conectáveis, existem grupos de Portas Não Conectáveis. Um grupo é um conjunto de portas para o qual pode-se enviar uma mensagem (fica transparente ao usuário se a mensagem está sendo enviada para um grupo ou para uma única porta). Os tipos de grupos são os mesmos que os conectores especiais: *broadcast*, que envia uma cópia a todas as portas do grupo, e *mailbox*, que a envia apenas para um membro do grupo (não existe o conceito de "grupo local", equivalente ao conector local).

2.3 Gerenciador de Processos

O Gerenciador é o responsável pela criação e gerenciamento dos processos distribuídos. Na medida do possível, ele estende as primitivas existentes no UNIX para funções semelhantes que operem sobre os processos da rede. Entre os serviços que oferece, podemos destacar:

- Criação de processos, com possibilidade de ajuste de um vasto número de atributos, tais como variáveis de ambiente, lista de argumentos, prioridade de execução, máscara de sinais, etc;
- Redirecionamento de descritores de arquivo dos processos para portas de comunicação;
- Envio de sinais a processos, grupos de processos ou membros de uma mesma sessão OMNI;
- Espera pela morte de um processo.

O Gerenciador de Processos é descrito em detalhes no capítulo 6.

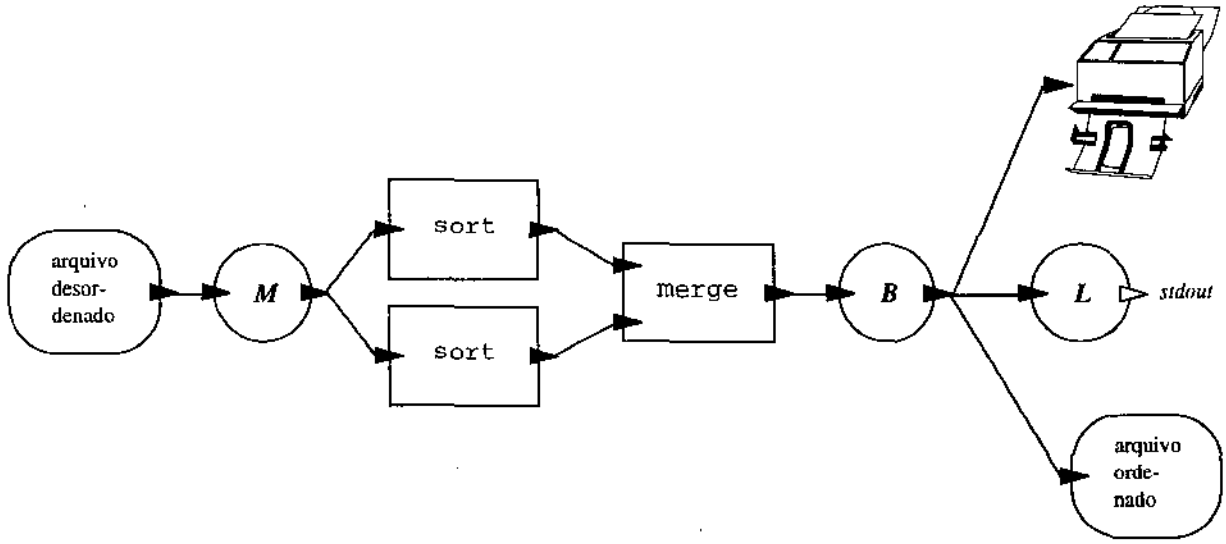
2.4 Usando o Sistema

Uma grande diversidade de aplicações podem ser implementadas usando o OMNI. Através dele, programas distribuídos podem criar processos em qualquer máquina da rede, que irão comunicar-se através de Portas de Comunicação, que podem ter sido cri-

adas tanto pelo próprio processo dono da porta quanto por seu pai, ao redirecionar alguns de seus descritores de arquivo para portas. O processo criador tem a possibilidade de monitorar a execução de seus filhos, sendo avisado quando eles terminarem. Se desejar, pode também mandar-lhes sinais, que podem abortar sua execução ou desviá-los para funções de tratamento. Como a conexão entre as portas pode ser feita por qualquer processo (que tenha as devidas permissões), abre-se a possibilidade de um processo criar vários outros e conectá-los conforme queira.

No exemplo mostrado na figura 2, um processo (não representado na figura) associa um arquivo desordenado a uma porta de comunicação, que é ligada a um conector especial de *mailbox*. A saída do conector é ligada às entradas de duas instâncias do programa `sort`. Este é o comando `sort` do UNIX, sem qualquer alteração, que teve sua entrada e saída padrão redirecionadas para Portas de Comunicação Conectáveis. As saídas dos programas `sort` são enviadas para `merge`, programado usando o OMNI, que as une. A saída de `merge` é por sua vez enviada a um conector de *broadcast*, que a replica para uma impressora, para um conector local e para uma porta associada a um arquivo em disco. O conector local, que herda os descritores de arquivo de seu criador, repassa o resultado para a saída padrão do processo que gerou a computação.

FIGURA 2 Diagrama do programa de ordenação distribuído `mergesort`.



Aplicações que utilizem o paradigma cliente-servidor podem também beneficiar-se do OMNI. Uma vez que o servidor fosse cadastrado no Servidor de Nomes, o cliente o consultaria e obteria a OMNIid do servidor, ficando transparente para ele sua localização. Em seguida conectaria suas portas e faria requisições. Caso a aplicação desejasse, a associação cliente-servidor não teria que ser feita pelo próprio cliente, mas por qualquer outro processo que possuísse a OMNIid da porta do cliente (por exemplo, o processo que disparou a execução do cliente).

Caso o servidor não estivesse executando, o próprio cliente poderia criá-lo através das primitivas de criação de processos e depois servir-se dele.

Capítulo 3

Outros Sistemas

O OMNI começou a ser elaborado por volta de fim de 1991/começo de 1992, com o objetivo de oferecer suporte às demais camadas do ambiente A_Hand [DL87a, DL87b]. Naquela época, assim como hoje, havia muita discussão em torno de sistemas distribuídos e procurou-se buscar inspiração para o OMNI nos principais sistemas existentes, como Chorus [RAA90], ISIS [Br93] ou MP [MD92], entre vários outros. Contudo, ao mesmo tempo que o OMNI ia sendo desenvolvido, cresceu o interesse da indústria de computação por esse área, que empreendeu grande esforço na criação de plataformas para o desenvolvimento de aplicações distribuídas. Isso levou à geração de sistemas como o DCE (*Distributed Computing Environment*) [DCE90], da OSF (*Open Software Foundation*), e à criação do modelo abstrato do ORB (*Object Request Broker*) [ORB91] pelo OMG (*Object Management Group*), transformado em seguida num modelo concreto com o advento do CORBA (*Common ORB Architecture*) [COR92a], também pelo OMG em conjunto com a X/Open.

OSF, OMG e X/Open juntos representam os interesses de empresas como IBM Corporation, Sun Microsystems, Inc., Hewlett-Packard Company, Digital Equipment Corporation, NCR Corportation, Siemens Nixdorf Informationssysteme AG, HyperDesk Corporation e Object Design, Inc., entre outras. Esses nomes dão uma amostra do peso que esses padrões exercerão sobre o mercado.

Boa parte da funcionalidade desses padrões emergentes é redundante com o OMNI (e/ou com o ambiente A_Hand). Se por um lado isso significa que o OMNI estava no caminho certo, significa também que, a menos que ele se adapte à utilização dentro ou em conjunto com esses ambientes, terá reduzida sua chance de sucesso entre os usuários. Ainda assim, grande parte das facilidades oferecidas pelo OMNI continuam sendo não oferecidas por esses padrões, principalmente no tocante a gerenciamento de processos.

3.1 Tipos de Suporte à Programação Distribuída

Podemos classificar as facilidades para suporte à programação distribuída atualmente existentes em três níveis:

1. Sistemas operacionais distribuídos;
2. Bibliotecas e programas construidos sobre o sistema operacional (seja ele distribuído ou não);
3. Linguagens ou ambientes orientados à programação distribuída.

No nível 1 podemos citar sistemas como Chorus [AGH89, RAA90], Mach [ABG86, JR86]. Sprite [DO87, OCD88] e V [Ch84, TLC85,Ch88]. Facilidades como OMNI, ISIS [BC90, CB92, Br93] e DCE [DCE90, Sh92, DCE93] encaixam-se no nível 2 e são mais comummente implementadas sobre sistemas operacionais centralizados, como o UNIX, por exemplo. Já no nível 3 podemos citar as três linguagens de programação do ambiente A_Hand (Cm [Fr91, Tl93, Gn94], *LegoShell* [Dr89] e CO^2 [FD91, Fd94]), além de outras, como CSP (*Comunicating Sequential Processes*) [Hr78], DP (*Distributed Processes*) [Hn78], MP [MDK91a, MD92] e Drawin [MDK91b], bem como facilidades como o CORBA [COR92a, COR92b].

Não discutiremos aqui maiores detalhes das facilidades de nível 3 existentes, pois isso já foi feito em [BLD88, Dr89, FD91, DGT93, Fd94, Gn94] com o objetivo de delinear o perfil das linguagens do ambiente A_Hand. Uma vez que o papel principal do OMNI é dar suporte a essas linguagens, as necessidades por elas apresentadas foram as maiores responsáveis por moldar seus serviços. Chegou-se à conclusão de que um sistema operacional distribuído seria a plataforma ideal para desenvolvê-las, mas que não se deseja abrir mão da portabilidade. Assim, o OMNI objetiva prover uma facilidade de nível 2 sobre o UNIX que crie a ilusão de que ele é um sistema operacional distribuído (ou seja, de que pertence ao nível 1), para que as linguagens do A_Hand (de nível 3) possam mais facilmente ser implementadas.

A brevidade impede que todas as fontes acima citadas possam ser discutidas em maiores detalhes, por isso nos limitaremos aqui a comentar apenas algumas das mais representativas. Todas entretanto, em maior ou menor grau, influenciaram o projeto atual do sistema.

Uma vez que o OMNI deseja criar a ilusão de um sistema operacional distribuído, é natural que sua maior fonte de inspiração sejam os SOs distribuídos já existentes. Dentre esses, podemos destacar o Chorus [AGH89, RAA90] como tendo exercido a maior influência. Um breve descrição do Chorus é apresentada na seção 3.2.

Dos sistemas de nível 2 existentes, salientamos o ISIS [BC90, CB92, Br93] que, apesar de prover facilidades mais básicas que as do OMNI, teve papel importante em expor as potenciais vulnerabilidades a falhas por ele apresentadas. O ISIS é brevemente descrito na seção 3.3. Por fim o DCE [DCE90, Sh92, DCE93] é apresentado na seção 3.4.

3.2 O Sistema Chorus

O Chorus em si não é um sistema operacional, mas sim um pequeno núcleo sobre o qual servidores de sistema podem ser construidos para formar um subsistema, que é visto pelo usuário como um sistema operacional. Um exemplo concreto disso é o Chorus/Mix, um subsistema UNIX construido sobre o Chorus. Com essa arquitetura, é possível inclusive ter vários subsistemas executando simultaneamente na mesma máquina. O núcleo Chorus ainda oferece facilidades para a programação de aplicações de tempo real.

As seguintes abstrações básicas são implementadas pelo núcleo:

- *Identificador único*: identifica os objetos do sistema unicamente no tempo e no espaço;
- *Ator*: unidade de alocação de recursos;
- *Thread*: unidade de execução seqüencial;
- *Mensagem*: unidade de comunicação;
- *Porta e Grupo de Portas*: unidade de endereçamento para o envio de mensagens e base de (re)configuração;
- *Região*: unidade de estruturação do espaço de endereçamento da memória virtual de um ator.

Três outras abstrações são gerenciadas cooperativamente pelo núcleo Chorus e pelos servidores de sistema:

- *Segmento*: unidade de encapsulamento de dados;
- *Capability*: unidade de controle de acesso;
- *Identificador de Proteção*: unidade de autenticação.

Um *ator* encapsula um conjunto de recursos:

- Um contexto de memória virtual, dividido em *regiões*, associadas a *segmentos* armazenados local ou remotamente;
- Um contexto de comunicação, composto de um conjunto de *portas*;
- Um contexto de execução, composto por um conjunto de *threads*.

Uma *thread* é um fluxo de execução. Ela está ligada a um e somente um ator e compartilha seus recursos com todas as outras *threads* daquele ator.

Uma *mensagem* é uma seqüência de *bytes* enviada a uma porta. No momento de sua criação, uma porta está atrelada a exatamente um ator e permite que *threads* dele recebam mensagens na porta. Uma porta pode migrar de um ator para outro. Qualquer *thread* que conheça a porta pode enviar mensagens a ela, mas somente as *threads* do ator associado à porta podem recebê-las. Portas podem ser dinamicamente agregadas em grupos, sendo que mensagens podem ser enviadas de forma transparente a uma porta do grupo ou a todas elas.

Atores, portas e grupos de portas todos têm *identificadores únicos* que são globais, independentes de localização e únicos no tempo e no espaço.

Segmentos são coleções de dados, gerenciados pelos servidores de sistema, que definem seu conteúdo. A localização dos dados providos pelo segmento nada tem a ver com a localização do usuário do segmento.

Duas facilidades são providas para a construção de mecanismos de controle de acesso e autenticação:

- Recursos podem ser identificados dentro de seus servidores por uma *chave* dependente do servidor. Essa chave, quando combinada com a identificação única do servidor, forma uma *capability* que pode ser usada para referenciar um recurso com segurança;
- Atores e portas recebem *identificadores de proteção* com os quais os núcleos carimbam todas as mensagens enviadas. Ao receber uma mensagem, um ator pode se valer desses identificadores para autenticação.

Vários conceitos do Chorus estão presentes no OMNI, como portas de comunicação, que podem ser agregadas em grupos, utilização de identificadores únicos e *capabilities*. As portas mais poderosas do OMNI, entretanto, são com conexão, ao passo que portas conectáveis não existem no Chorus. A facilidade de criação de processos remotos do OMNI é relativamente tão poderosa quanto a do subsistema Chrorus/Mix. A seu favor, o OMNI tem também o servidor de Nomes, que provê uma grau de transparência não existente no Chrous.

O OMNI é também muito mais portável que o Chorus, uma vez que pode operar sobre qualquer UNIX. Não faz sentido portar o Chorus ou Chorus/Mix para outro sistema operacional, uma vez que ele *é* o sistema operacional. Sua portabilidade (que, aliás, é grande), aplica-se apenas a plataformas de *hardware* e não de *software*.

3.3 O Sistema ISIS

O sistema ISIS é uma ferramenta simples, poderosa e elegante que provê mecanismos de comunicação confiável entre grupos de processos. Através da utilização do modelo por ele proposto, é possível ao programador comum projetar aplicações distribuídas altamente tolerantes a falhas, pois o sistema foi projetado tendo tolerância a falhas como máxima prioridade.

Central ao ISIS é o modelo de *sincronia virtual* (*virtual synchrony*), que determina o comportamento de grupos processos distribuídos comunicando-se através de *multicasts*, que são mensagens enviadas simultaneamente a todos os membros do grupo. Sincronia virtual é um relaxamento de um modelo mais robusto, chamado de *sincronia forte* (*close synchrony*). A sincronia forte garante as seguintes propriedades:

- A execução de um processo consiste de uma seqüência de eventos, que podem ser computações internas, transmissão de mensagens, entrega de mensagens ou mudanças na composição de grupos que cria ou aos quais se incorpora;
- A execução global do sistema consiste de um conjunto de execuções de processos. No nível global, mensagens podem ser vistas como *multicasts* a grupos de processos;
- Quaisquer dois processos que recebam os mesmos *multicasts* ou observem as mesmas mudanças na composição de um grupo enxergam os correspondentes eventos locais na mesma ordem relativa;
- Um *multicast* a um grupo de processos é entregue a todos os membros. Os eventos de envio e entrega da mensagem são considerados como um único evento instantâneo.

Sincronia forte é uma garantia muito poderosa, eliminando problemas derivados da falta de confiabilidade na comunicação, mudança na composição dos membros de um grupo, ordem de entrega das mensagens, transições de estado dos processos e não atomicidade na presença de falhas. Entretanto, além de problemas teóricos (os quais não discutiremos aqui) que impedem sua implementação exata na presença de falhas, o custo de manter sincronia forte é altíssimo, pois implica que os processos progridam em passos sincronizados (*lock step*).

Assim sendo, o ISIS relaxa esse conceito para o de *sincronia virtual*, onde as interações entre os processos são assíncronas. Após enviar um *multicast*, um processo pode continuar executando sem ter que esperar por sua entrega aos membros do grupo. Dessa forma o sistema de comunicação se comporta como um *buffer* finito, bloqueando o remetente somente quando a taxa de produção de suas mensagens é maior do que a de consumo, ou quando ele tem que aguardar por alguma resposta. As demais propriedades da sincronia forte são preservadas, a menos que relaxadas pelo usuário.

Graças ao modelo de sincronia virtual, o programador pode fazer uma série de suposições sobre o estado e o comportamento de sua aplicação distribuída, cuja validade não teria como ser garantida sem as propriedades oferecidas pelo modelo. Isso simplifica grandemente o projeto da aplicação e lhe permite alcançar um alto grau de tolerância a falhas.

Além de facilidades de comunicação ponto-a-ponto, o sistema provê dois tipos de *multicast*, que diferem no tipo de ordenação que provêem entre as mensagens:

- ABCAST, que garante ordenação total (mais custoso),
- CBCAST, que provê ordenação causal (mais eficiente).

A ordenação causal é mais "fraca" que a ordenação total, pois garante apenas que uma mensagem nunca é entregue a um membro do grupo antes de outras das quais ela dependa potencialmente. Por exemplo, se A envia a mensagem m_1 para B e C e, após receber m_1, B envia m_2 para A e C, então C deve receber m_1 antes de m_2, pois pode ser que m_2 dependa de m_1. Por outro lado, se A envia m_1 ao mesmo tempo que B envia m_2,

C pode recebê-las em qualquer ordem, pois não existe relação de dependência causal entre elas.

Sobre a base oferecida pelo ISIS, outros sistemas foram implementados, como o Gerenciador de Recursos (ISIS *Resource Manager*), que monitora a carga nas máquinas da rede, implementando a abstração de um conjunto de recursos (tipicamente processadores) aos quais são alocadas tarefas (processos) de forma transparente ao usuário. Isso permite um melhor aproveitamento de poder computacional das máquinas ociosas da rede.

O OMNI é um sistema de nível um pouco mais alto do que o ISIS, cuja função principal é apenas prover um mecanismo de *multicast* tolerante a falhas. O ISIS não possui serviços equivalentes aos do Servidor de Nomes do OMNI, por exemplo. O Gerenciador de Recursos sobre ele implementado provê uma maior transparência do que o Gerenciador de Processos do OMNI, uma vez que decide por si só onde um processo irá executar, sendo que no OMNI isso é especificado pelo usuário. Deve-se ter em mente, entretanto, que os critérios usados pelo ISIS para a escolha da máquina que executará o processo são baseados quase que exclusivamente na carga das máquinas. Apesar desse critério ser bom para o ISIS, onde a comunicação entre os processos consiste-se primordialmente de *multicasts*, o mesmo não se pode dizer do OMNI, onde localização de arquivos e a intensidade de comunicação entre pares de processos pode ser mais importante do que a simples carga da máquina (processos fazendo acesso intenso a um arquivo devem executar próximos ao arquivo, processos que se comunicam intensamente devem estar próximos um do outro). Devido a esse fato, o monitoramento da carga das máquinas foi deixado de fora do OMNI, com planos de implementá-lo usando o próprio sistema, de forma a permitir que o usuário especifique critérios adequados para a escolha automática das máquinas.

No tocante à comunicação entre os processos, as primitivas ISIS são orientadas a prover alto grau de tolerância a falhas, através do modelo de sincronia virtual. O OMNI, por não seguir esse modelo, não possui o mesmo grau de tolerância, porém não necessita incluir em sua comunicação a carga adicional (*overhead*) por ele imposto. Além disso, as primitivas OMNI foram projetadas para permitir seu uso mesmo por processos comuns (isto é, que não foram implementados usando o OMNI), através do redirecionamento de descritores padrão para portas de comunicação, ao passo que isso não é possível no ISIS.

3.4 O Ambiente DCE

O Ambiente para Computação Distribuída (*Distributed Computing Environment* ou DCE) da OSF (*Open Software Foundation*) é um conjunto integrado de serviços de rede que oferecem suporte ao desenvolvimento, uso e manutenção de aplicações distribuídas. Tais serviços estão organizados em duas categorias:

- Serviços distribuídos fundamentais, que provêem ferramentas para desenvolvedores de *software* criarem os serviços necessários ao usuário final de computação distribuída. Eles incluem:

  - Chamada Remota de Procedimento (RPC),
  - Serviço de Nomeação,
  - Serviço de Tempo,
  - Serviço de Segurança,
  - Serviço de *Threads*,
- Serviços de Compartilhamento de dados, que provêem aos usuários finais recursos construidos sobre os Serviços Fundamentais. Eles incluem:
  - Sistema de Arquivos Distribuído,
  - Suporte a computadores Sem Disco,
  - Serviço de suporte a arquivos e impressoras MS-DOS.

3.4.1 Serviços Distribuídos Fundamentais

Os componentes fundamentais do DCE, que formam a base sobre a qual aplicações distribuídas podem ser construidas, são brevemente descritos abaixo:

3.4.1.1 Chamada Remota de Procedimento (RPC)

Esta facilidade permite que o programador invoque procedimentos dentro de seu programa que serão de fato executados em outro processo. RPC mascara a representação dos dados em diferentes máquinas, permitindo a cooperação entre sistemas heterogêneos.

Os recursos de DCE RPC são divididos em dois componentes: uma facilidade para a invocação de RPC, que provê simplicidade, desempenho, portabilidade, independência de rede e de protocolo e segurança (veja seção 3.4.1.5); e um compilador, que converte descrições de alto nível das interfaces dos procedimentos remotos em código fonte portável em C.

3.4.1.2 Serviço de Nomeação Distribuído (*Distributed Naming Service*)

Este serviço provê um modelo único de nomeação em todo o ambiente distribuído. Este modelo permite identificar por nome recursos tais como servidores, arquivos, discos ou filas de impressão, e obter acesso a eles sem conhecer sua localização na rede. Além disso, os usuários podem continuar referenciando um recurso pelo mesmo nome, mesmo quando uma característica do recurso, como seu endereço na rede, por exemplo, é alterada.

O Serviço de Nomeação suporta o padrão de nomeação X.500, da ISO e CCITT, o que possibilita a interoperabilidade com um serviço global de nomeação. Além disso, são também oferecidas facilidades de *replicação*, o que provê melhor tempo de resposta e maior tolerância a falhas; *caching*, que proporciona maior desempenho; *segurança*, devido à integração com os Serviços de Segurança (veja seção 3.4.1.5); *capacidade de crescimento* (*scalability*), sendo capaz de acomodar tanto grandes quanto pequenas redes; e *independência de protocolo* de transporte, por ser implementado sobre DCE RPC (veja seção 3.4.1.1).

3.4.1.3 Serviço de Tempo

Provê um serviço de sincronização entre os relógios das máquinas da rede com relação a um padrão largamente reconhecido. Este serviço é tolerante a falhas, de fácil gerenciamento e garante a monotonicidade dos relógios, jamais ajustando nenhum deles para trás (atrasando-o), o que poderia causar um comportamento incorreto em certas aplicações.

3.4.1.4 Serviço de *Threads*

Permite a criação de mais de um fluxo de execução dentro do mesmo processo, oferecendo também mecanismos de sincronização entre eles. Isso permite que o poder computacional de máquinas multiprocessadas seja melhor utilizado, possibilitando também que fluxos independentes do mesmo processo continuem executando mesmo que outros estejam bloqueados. Essa facilidade é especialmente útil para servidores que atendem múltiplos clientes ao mesmo tempo.

3.4.1.5 Serviço de Segurança

Este serviço está dividido em três componentes: autenticação, autorização e gerenciamento de contas de usuário. Estes serviços estão disponíveis através de facilidades de comunicação que garantem a privacidade e a integridade dos dados.

O serviço de autenticação baseia-se no sistema Kerberos, desenvolvido no Projeto Athena do MIT, que valida a identidade de um usuário ou serviço, impedindo requisições fraudulentas. Depois de autenticados, os usuários devem receber autorização para utilizarem recursos, como arquivos, por exemplo. O DCE provê as ferramentas necessárias para uma aplicação decidir se um usuário tem acesso a um determinado recurso ou não.

DCE oferece também facilidades para o gerenciamento de contas de usuário num ambiente distribuído e heterogêneo, garantindo nomes e senhas únicas entre os sistemas e serviços da rede, exatidão e consistência dessa informação em todas as máquinas e segurança no momento que são introduzidas alterações.

3.4.2 Serviços de Compartilhamento de Dados

Estes são serviços construidos sobre os serviços fundamentais. Uma vez integrados ao sistema operacional, eles fornecem facilidades chave para o usuário. São eles:

3.4.2.1 Sistema de Arquivos Distribuído

Este é o componente chave do DCE para o compartilhamento de informação. Ele provê ao usuário um espaço uniforme de nomes, transparência quanto à localização dos arquivos e alta disponibilidade, tendo um excelente desempenho mesmo quando grandes distâncias e muitos usuários estão envolvidos.

Sua confiabilidade é garantida através de um sistema de arquivos físico baseado em registros (*logs*) que permite uma rápida recuperação após falhas do servidor. Arquivos e diretórios são replicados de forma invisível em múltiplas máquinas, provendo acesso confiável e alta disponibilidade, mesmo na presença de falhas do servidor.

O Sistema de Arquivos Distribuídos do DCE oferece interoperabilidade com relação a outros sistemas de arquivos, incluindo o NFS (*Network File System*) da Sun.

3.4.2.2 Suporte a Computadores Sem Disco

O Sistema de Arquivos do DCE provê suporte a computadores que não tenham disco próprio, permitindo uma diminuição dos custos do *hardware* e mantendo alto o desempenho da rede.

3.4.2.3 Serviço de Integração a Computadores Pessoais

Permite que computadores pessoais compartilhem recursos como arquivos, periféricos e aplicações num ambiente distribuído.

3.4.3 Comparação com o Sistema OMNI

O DCE oferece vários dos serviços do OMNI, com a vantagem de ser um padrão apoiado pela maioria das empresas comprometidas com a filosofia de Sistemas Abertos. Ele não oferece, entretanto, nenhum mecanismo de criação de processos equivalente ao Gerenciador de Processos do OMNI.

O Serviço de Nomeação do DCE armazena seus dados de forma centralizada e oferece tolerância a falhas através de replicação em várias máquinas. O Servidor de Nomes do OMNI, por sua vez, mantém os dados distribuídos e a tolerância a falhas decorre de sua arquitetura: uma vez que os registros dos nomes são armazenados na máquina onde reside a entidade possuidora do nome, caso a máquina falhe, tanto a entidade quanto seu registro no Servidor são perdidos conjuntamente, sem que seja necessário sua remoção explícita da base de dados.

A comunicação no DCE é baseada quase que exclusivamente em RPC, que é ótimo para o modelo cliente-servidor, mas nem sempre se encaixa a outros modelos. As portas de comunicação do OMNI permitem que um fluxo de dados flua de um produtor para o consumidor de forma mais natural que RPC. Portas permitem também uma melhor integração de processos comuns (que não usam OMNI ou DCE) em computações distribuídas, através do redirecionamento de seus descritores de arquivo para portas.

Abstrações como conectores especiais ou grupos de portas também não têm nenhum equivalente no DCE, o que torna a comunicação entre grupos de processos mais complicada, principalmente quando feita por RPC, que é um modelo de comunicação ponto-a-ponto.

Capítulo 4

O Servidor de Nomes

O Servidor de Nomes provê transparência quanto à localização das entidades do sistema. Ele associa nomes simbólicos (seqüências de caracteres terminadas por um caractere nulo) a identificações OMNI (abreviadamente chamadas de OMNIids), que são estruturas de dados usadas para referenciar entidades OMNI espalhadas pela rede. Ou seja, quando uma entidade é criada (por exemplo, quando o usuário cria uma porta de comunicação), o sistema automaticamente cria para essa entidade uma OMNIid. O usuário tem então a opção de associar um nome simbólico à OMNIid, registrando-a no Servidor de Nomes. Um outro processo, potencialmente remoto, pode então recuperar a OMNIid fornecendo o nome simbólico ao Servidor de Nomes. Uma vez de posse da OMNIid, ele pode referenciar a entidade por ela identificada.

4.1 Definindo a OMNIid

A OMNIid é uma estrutura de dados utilizada pelas primitivas do sistema para referenciar qualquer entidade OMNI. Quando um usuário pede para criar uma entidade (por exemplo, quando ele pede para criar um processo remoto), o sistema cria a entidade e atribui a ela uma OMNIid, que é retornada ao usuário. Quando ele quiser novamente referenciar a entidade (por exemplo, se ele agora quiser matar o processo), ele deve fornecer a OMNIid à primitiva OMNI apropriada. A OMNIid pode ser divulgada a outros processos e eles poderão também referenciar a mesma entidade, onde quer que estejam.

A OMNIid identifica uma entidade unicamente no tempo e no espaço. Ou seja: se uma entidade possui uma OMNIid com um certo valor, é garantido que nunca existiu ou existirá outra OMNIid com o mesmo valor em nenhum ponto da rede, mesmo que a entidade em questão deixe de existir. Isso é necessário num sistema distribuído, pois caso contrário corre-se o risco de referenciar por engano uma entidade diferente com a mesma OMNIid. Imagine o seguinte cenário:

- O processo A cria a porta P_a com OMNIid I;
- O processo B obtém I;
- A porta P_a deixa de existir, por qualquer que seja o motivo;
- O processo C cria uma porta P_c com identificação I, igual à identificação anteriormente dada a P_a;
- O processo B referencia a porta cuja identificação é I. Ele pensará estar referenciando P_a, mas estará na verdade referenciando P_c.

A fim de atender os requisitos de unicidade espacial e temporal, a OMNIid incorpora uma *identificação única* (*unique id* ou uqid) que possui os seguintes campos:

1. Identificação da máquina onde a entidade foi originalmente criada. Para este campo utiliza-se atualmente o endereço Internet [Co88] da máquina, que a identifica unicamente no mundo inteiro (no caso da máquina possuir mais de um endereço, escolhe-se aquele com menor valor numérico);
2. Identificação UNIX (veja `getpid(2V)`, em [SRM90]) do processo que criou (ou onde reside) a entidade;
3. Instante de inicialização do processo que criou (ou no qual reside) a entidade;
4. Número seqüencial crescente, relativo ao processo que criou (ou onde reside) a entidade.

O campo 1 garante uma identificação única de cada máquina em todo o planeta. Dentro da máquina, o campo 2 identifica qual o processo criador. Contudo, como as identificações UNIX para os processos se repetem ao longo do tempo, o campo 3 serve para identificar no tempo qual o processo em questão. Como um processo pode criar muitas entidades, o campo 4 identifica qual delas se deseja referenciar.

A OMNIid possui ainda outros campos, inseridos por questão de eficiência ou segurança:

- *Capability*. Este é um número aleatório, usado para dificultar a construção de uma OMNIid por parte de algum usuário que tente referenciar uma entidade para a qual não tenha permissão de acesso;
- Tipo da entidade. É um número que informa qual o tipo da entidade (processo, porta de comunicação, grupo de portas, etc) à qual se refere a identificação;
- Informação específica. Para cada diferente tipo de entidade existem informações específicas para a manipulação daquele tipo particular de entidade (em C, este campo é implementado como uma `union`).

4.2 Associando Nomes Simbólicos a OMNIids

Para referenciar uma entidade OMNI, é preciso conhecer sua OMNIid. É necessário então que as entidades tenham uma maneira de divulgar suas OMNIids aos possíveis interessados. Por exemplo: um processo servidor poderia desejar divulgar a OMNIid de sua porta de comunicação, a fim de permitir que seus clientes tomem conhecimento

dela para poderem enviar requisições de serviço. A função do Servidor de Nomes é exatamente permitir a associação de nomes simbólicos, que são seqüências de caracteres, a OMNIids. O servidor poderia então registrar-se junto ao Servidor de Nomes. Por sua vez, o cliente, sabendo de antemão qual o nome do servidor, poderia consultar o Servidor de Nomes e obter a OMNIid. Note que quem determina a OMNIid de uma entidade é o Sistema OMNI, mas quem determina o nome a ela associado é o usuário que a criou.

Apesar do Servidor de Nomes ter sido projetado para ser utilizado pelas camadas superiores do OMNI, ele é genérico e pode ser utilizado para associar qualquer informação a um nome simbólico, não apenas OMNIids. O Servidor não interpreta de forma alguma os dados associados ao nome e não conhece o conceito de OMNIid. Ele recebe a OMNIid das camadas superiores e apenas a armazena associada ao nome. Mais tarde, quando o nome é consultado, ele apenas retorna a OMNIid. É perfeitamente possível imaginar aplicações que utilizem o Servidor de Nomes diretamente, sem fazer uso das demais camadas do OMNI, registrando nele não OMNIids, mas qualquer outra informação pertinente à essa aplicação específica.

4.3 Descrição funcional do Servidor de Nomes

O Servidor de Nomes é um programa distribuído constituído por vários *daemons*, um em cada máquina. Cada *daemon* armazena os registros cadastrados por processos residentes naquela máquina. Os principais serviços que o Servidor de Nomes oferece são:

- `ns_init`: inicializa o processo para que possa usar o Servidor de Nomes;
- `ns_putName`: cadastra um par nome-informação[1];
- `ns_getInfo`: dado um nome, recupera a primeira informação associada que encontrar;
- `ns_getAllInfo`: dado um nome, recupera todas as informações a ele associadas;
- `ns_removeName`: remove um par nome-informação;
- `ns_removeProcNames`: remove todos os pares nome-informação cadastrados por um processo específico;
- `ns_removeUsrNames`: remove todos os pares cadastrados por um usuário específico;
- `ns_purgeName`: retira do *cache* de nomes externos todas as entradas associadas a um certo nome;
- `ns_purgeCache`: retira do *cache* de nomes externos os nomes registrados por processos ou usuários específicos.

1. No caso do Sistema OMNI, essa "informação" é sempre uma OMNIid. Aplicações que utilizem o Servidor de Nomes diretamente podem armazenar outros tipos de informação.

4.3.1 Contextos de Nomes

Uma vez que o Servidor de Nomes pode ser utilizado por uma variada gama de aplicações e não apenas pelas camadas superiores do OMNI, faz-se necessário que nomes registrados por uma aplicação não se confundam com aqueles registrados por outras. Assim sendo, O Servidor suporta o conceito de *contextos* de nomes. Um contexto é um espaço de nomes independente. Dois nomes iguais em contextos diferentes não se confundem. Um nome pode ser único somente em relação a seu próprio contexto.

Sempre que é feita uma operação sobre um nome, seja de inserção, busca ou remoção, o usuário deve especificar a qual contexto se refere. As camadas superiores do OMNI possuem um contexto próprio que apenas elas irão utilizar para registrar os nomes das entidades OMNI.

4.3.2 Domínios do Servidor de Nomes

O Servidor de Nomes é um programa distribuído que executa as buscas por nomes em conjunto fixo de máquinas, configurável pelo administrador do sistema. A esse conjunto damos o nome de *domínio* do Servidor de Nomes. Ou seja, se um nome é cadastrado por um processo executando em uma das máquinas do domínio, ele pode ser consultado por qualquer máquina daquele domínio. Se um processo quiser consultar um nome registrado num domínio diferente do da sua própria máquina, ele tem que explicitamente fornecer o nome (ou endereço) de uma máquina qualquer do domínio em que se encontra o nome antes de requisitar a consulta, para que essa máquina efetue a busca naquele domínio.

De modo a facilitar o trabalho de administração da rede, recomenda-se que o domínio do Servidor de Nomes coincida com o domínio NIS (*Network Information Service*) [SNA90] e/ou com o domínio Internet ao qual pertence a máquina.

4.3.3 Atributos de um Nome Simbólico

Todos os nomes simbólicos possuem os seguintes atributos, que podem ser especificados no momento em que o nome é cadastrado no Servidor:

- Visibilidade: global ou local. Um nome com visibilidade local só pode ser referenciado por processos residentes na mesma máquina em que o nome foi registrado. Visibilidade global permite que o nome seja referenciado de qualquer parte da rede;
- Unicidade: único global, único local ou não necessariamente único. Se um nome é único global, não existe nenhum outro na rede igual a ele. Se é único local, garante-se apenas que não há nenhum outro igual na mesma máquina, e se é não necessariamente único, nada se pode afirmar sobre a existência de outros nomes iguais (note que, conforme dito anteriormente, o conceito de unicidade é referente apenas ao contexto e ao domínio onde o nome foi cadastrado);
- Dono: identificação do usuário que cadastrou o nome;
- Grupo: grupo efetivo de usuários ao qual pertence o processo no instante do cadastramento do nome;

- Domínio: identificação do domínio do Servidor de Nomes em que o nome foi cadastrado;
- Permissões de acesso: conforme dito na seção seção 4.1, a OMNIid de uma entidade possui uma *capability* que permite a quem a conheça a capacidade de manipular a entidade. Por exemplo, se um processo sabe a OMNIid de uma Porta de Comunicação, ele pode conectar-se àquela porta. Sendo assim, faz-se necessário prover um mecanismo de controle de acesso às informações registradas no Servidor. A todo nome são associados *bits* de permissão de acesso para o dono, grupo e domínio, bem como para outros usuários que não estão em nenhuma das três primeiras categorias. De forma semelhante a arquivos no UNIX, pode-se ligar ou desligar cada um desses quatro *bits,* concedendo ou negando o acesso às informações associadas ao nome para cada um dos quatro conjuntos de usuários;
- Tipo: número inteiro associado a cada nome que é usado como indicação do tipo de informação associada ao nome. No caso do OMNI, por exemplo, pode ser usado para dizer se o nome se refere a uma porta, um processo, um grupo de portas, etc. O Servidor não interpreta o valor desse campo, a não ser quando ele é usado para limitar uma consulta (pode-se, por exemplo, pedir para buscar um registro de um nome com tipo "porta", ignorando-se os demais).

4.4 Especificação Interna do Servidor de Nomes

O Servidor de Nomes é composto de vários *daemons*, um executando em cada máquina do seu domínio, e por uma biblioteca de funções, usada pelos clientes para fazer acesso aos serviços. Quando um nome é cadastrado, ele fica armazenado no *daemon* da máquina onde reside o processo que o cadastrou. Quando é feita uma consulta, procura-se o nome primeiramente no *daemon* local; caso não seja achado procura-se no *cache* de nomes externos mantido pelo *daemon* local; caso ainda não seja achado, faz-se um *broadcast* no domínio para descobrir se algum *daemon* tem aquele nome registrado.

Detalharemos melhor agora esses algoritmos e as estruturas de dados utilizadas para implementá-los.

4.4.1 Estruturas de Dados do Servidor de Nomes

O *daemon* possui as seguintes estruturas:

- Tabela de Nomes Locais (TNL): Possui os pares nome-informação registrados por processos locais (ou seja, residentes na mesma máquina que o *daemon*);
- *Cache* de Nomes Externos (CNE): Possui o registro (par nome-informação) dos nomes externos ao *daemon* que mais recentemente foram referenciados por processos locais. A consistência do *cache* é deixada a cargo do usuário, conforme explicado na seção 4.4.5;
- Tabela de Nomes Únicos Globais Externos (TNUGE): esta tabela tem a seguinte propriedade: se um nome único global existe no domínio, então ele tem pelo menos uma entrada na tabela (com exceção daqueles que já constam na TNL). Nada se pode afirmar caso o nome não exista. É também possível que o mesmo nome possua

mais de uma entrada, caso em que no máximo uma das entradas está correta (a informação obsoleta é removida de tempos em tempos por um procedimento de coleta de lixo descrito na seção 4.4.7). Esta tabela contém apenas os nomes e sua localização, mas não a informação associada ao nome. Ao contrário do que possa parecer à primeira vista, o propósito desta tabela é acelerar a inserção de nomes únicos locais ou não necessariamente únicos.

4.4.2 Inserção de um Nome Único Global

Os seguintes passos são efetuados para inserir um nome único global no domínio do Servidor de Nomes:

- Procura-se o nome na TNL. Caso ele exista, a inserção falha;
- Procura-se o nome na TNUGE. Caso ele conste na tabela, para cada entrada com aquele nome entra-se em contato com o *daemon* que supostamente contém aquele registro e pede-se a confirmação da existência do nome. Se o *daemon* confirmar, a inserção falha. Caso contrário, retira-se a(s) entrada(s) inconsistente(s) da TNUGE;
- Vasculha-se o CNE à procura do nome. Se forem encontradas entradas referentes ao nome onde ele consta como único global, essa entrada é removida do *cache*. Se forem achadas entradas onde ele aparece como único local de alguma máquina ou não necessariamente único, entra-se em contato com o(s) respectivos *daemon(s)* a fim de confirmar sua(s) existência(s). Se a existência não é confirmada, remove-se aquela entrada do *cache*. Caso contrário, a entrada não é removida e a inserção falha;
- Efetua-se um *broadcast* no domínio do Servidor com a mensagem: "a máquina M com prioridade P_m deseja registrar o nome N como único global no contexto C". Cada *daemon* do domínio recebe a mensagem e procura o nome em sua TNL. Caso o encontre, envia uma resposta negando o pedido, juntamente com informação sobre o nome para ser colocada no CNE da máquina local. Caso contrário, insere o nome em sua TNUGE e manda uma resposta aceitando o pedido;
- A cada *daemon* é associada uma prioridade diferente e fixa em relação aos outros. No caso de dois *daemons* A e B, onde a prioridade de A é maior que a de B, tomarem a iniciativa de inserir o mesmo nome simultaneamente, é certo que em um dado instante o *broadcast* enviado por um irá alcançar o outro. Nesse caso, quando A receber o pedido de B ele irá negá-lo, alegando que está tentando criar o mesmo nome e possui prioridade maior. O *daemon* B, por sua vez, quando receber o pedido de A irá concedê-lo e desistirá de inserir o nome (se ainda não tinha desistido), pois reconhecerá que tem prioridade menor;
- Se todos os *daemons* aceitarem a criação do nome ele é inserido na TNL e o algoritmo de inserção termina com sucesso.

4.4.3 Inserção de um Nome Único Local ou Não Necessariamente Único

A inserção de um nome único global, conforme descrita na seção anterior, é uma operação cara, pois envolve um *broadcast* no domínio. O *broadcast* é necessário para garantir que nenhuma outra máquina tenha o nome que se deseja criar como único. No entanto, a criação de um nome único local ou não necessariamente único não deveria

ser cara, pois não se deseja ter a propriedade de unicidade no domínio. Contudo, a criação de tal nome sem uma maneira de garantir que não exista nenhum outro nome único global idêntico poderia violar a unicidade *deste* nome. A TNUGE existe exatamente para permitir que um nome único local ou não necessariamente único possa ser criado sem que seja feito um *broadcast* no domínio.

Os passos para a criação de um nome único local são:

1. Procura-se o nome na TNL. Se ele existir, a inserção falha;
2. Procura-se o nome na TNUGE. Caso ele conste na tabela, para cada entrada com aquele nome entra-se em contato com o *daemon* que supostamente contém aquele registro e pede-se a confirmação da existência do nome. Se o *daemon* confirmar, a inserção falha. Caso contrário, retira-se a(s) entrada(s) inconsistente(s) da TNUGE;
3. Insere-se o nome na TNL.

Para criar um nome não necessariamente único os passos são os mesmos, com a diferença que o passo 1 falha apenas se já existir algum nome único local ou único global idêntico.

4.4.4 Busca de um Nome no Servidor

Quando um cliente faz uma consulta ao Servidor de Nomes, os seguintes passos são executados:

- O *daemon* local procura o nome requisitado em sua TNL;
- Caso não o encontre, o procura no CNE;
- Se não o achar, procura na TNUGE. Caso ele conste dessa tabela, entra em contato com o *daemon* responsável por seu armazenamento e requisita a informação associada ao nome (lembre-se que a TNUGE guarda apenas o nome e sua localização, mas não a informação a ele associada. Além disso, a entrada na TNUGE pode ser inconsistente);
- Caso ainda assim não o encontre, efetua um *broadcast* no domínio do Servidor de Nomes requisitando o nome. Se alguma máquina responder afirmativamente, o nome é colocado no CNE e retornado ao cliente.

A menos que o nome seja único global, é possível que exista mais de uma informação associada ao mesmo nome. Os passos acima são executados quando o cliente está interessado apenas em *uma* dessas informações, que será a primeira que o Servidor conseguir encontrar. É possível também requisitar *todas* as informações associadas a um nome. Nesse caso, o *daemon* sempre executa um *broadcast* e espera pela resposta de todas as máquinas, retornando ao cliente todas as informações de todas as instâncias do nome.

4.4.5 Remoção de um Nome do Servidor

O usuário pode remover do Servidor uma entrada para a qual não tenha mais utilidade. Quando um nome é removido, a única ação efetuada é retirá-lo da TNL. O *daemon* local não informa nenhum outro sobre a remoção. Isso faz com que a operação de remo-

ção seja barata, pois não envolve nenhuma outra máquina além da local, mas abre possibilidade para que as TNUGEs e CNEs das outras máquinas fiquem inconsistentes.

Conforme explicado anteriormente, a inconsistência na TNUGE não constitui nenhum problema, pois os algoritmos sempre confirmam a consistência de uma entrada antes de utilizá-la. Uma entrada inconsistente na TNUGE acaba sempre sendo removida quando se tenta criar um nome idêntico ou quando é feita uma coleta de lixo (veja seção seção 4.4.7).

Já a consistência do CNE é deixada a cargo do usuário. O Servidor de Nomes não tenta manter consistentes os *caches* porque a tentativa seria cara (pois envolveria um *broadcast*) e de qualquer forma de nada adiantaria. Imagine o seguinte cenário:

1. Um cliente consulta o Servidor com respeito a um nome *N*;
2. O Servidor encontra *N* em alguma máquina remota, o coloca em seu CNE e o devolve ao cliente;
3. *N* é removido do Servidor de Nomes;
4. O cliente tenta usar N.

Note que mesmo se, no passo 3, o nome *N* fosse instantaneamente (ou atomicamente) retirado da máquina onde reside e de todos os *caches* onde estava armazenado, ainda assim o cliente tentaria fazer acesso a um nome que não mais existe no passo 4. Ora, se o cliente ia mesmo ter que se preocupar com o caso em que a informação que possui não é mais válida, nada mais razoável então do que deixar a consistência do *cache* a seu encargo. Quando o cliente tentar usar a informação e não conseguir por ela estar obsoleta, ele deve avisar o Servidor de Nomes para remover aquela entrada do CNE (apenas o *cache* do *daemon* local é atualizado). Note que, no caso das camadas superiores do OMNI, a maneira como a OMNIid é construida (veja seção seção 4.1) tem papel essencial em permitir que seja possível identificar se uma entidade é realmente aquela que se desejava referenciar, pois a OMNIid é única no tempo e no espaço para cada entidade.

O CNE tem um tamanho fixo (configurável pelo administrador) e usa uma política de LRU (*Least Recently Used*). Logo, mesmo que uma entrada nunca seja explicitamente retirada do *cache*, ela acaba inevitavelmente sendo descartada quando se tornar muito velha.

4.4.6 Inicialização[1] do Servidor de Nomes

Quando um *daemon* do Servidor de Nomes começa a executar, sua TNG e seu CNE estão vazios, mas ele precisa ser capaz de saber quais são as máquinas pertencentes ao seu domínio (para poder efetuar *broadcasts*) e quais são os nomes únicos globais que já existem, para poder construir sua TNUGE.

1. A palavra "inicialização" e o verbo "inicializar" são derivados do Inglês *initialize* e não existem oficialmente na língua portuguesa, mas estão sendo empregados aqui assim mesmo, por serem um jargão amplamente utilizado em computação e por não haver em Português nenhum termo com o mesmo significado.

A informação de quais máquinas fazem parte do domínio encontra-se num arquivo de configuração, que pode ser distribuído na rede usando NFS ou replicado em cada máquina. Esse arquivo contém todas as máquinas do domínio, mas não necessariamente todas elas estão ativas num dado instante. O Servidor envia um *broadcast* para todas informando que está se inicializando e, a partir das respostas que recebe, monta uma lista das máquinas ativas no domínio. Note que o *daemon* foi incluído na lista de máquinas ativas dos outros *daemons* por ter feito esse *broadcast*.

Em seguida o *daemon* envia uma mensagem para qualquer uma das máquinas do domínio e pede que ela lhe envie sua TNUGE, acrescida de todos os nomes únicos globais que constarem em sua TNL.

Durante o período de inicialização, o *daemon* responde com uma mensagem de erro a qualquer cliente seu que tente registrar um nome nele. Além disso, se ele é inquirido por outro *daemon* sobre a existência de algum nome, ele envia uma resposta dizendo que não possui o nome em questão. Se essa consulta for uma tentativa de criar um nome único global, o *daemon* inclui o nome em sua TNUGE (mesmo se ela ainda estiver vazia). Mais tarde, quando obtiver o conteúdo da TNUGE de outra máquina, ele irá fazer a união da sua TNUGE com a que acabou de receber.

4.4.7 Coleta de Lixo na TNUGE

A TNUGE não tenta se manter consistente com os nomes únicos globais que de fato existem em cada *daemon* do domínio do Servidor de Nomes. Ao invés disso, ela possui a propriedade mais fraca de garantir apenas que se o nome existe então ele está na tabela, mas não a recíproca.

Tentar garantir total consistência da TNUGE seria extremamente caro, pois implicaria em seqüenciar no domínio inteiro as criações de nomes únicos globais. Ou seja: enquanto um *daemon* está inserindo um nome único global, nenhum outro pode também estar fazendo isso. Se não for imposta essa forte restrição, corre-se o risco de duas máquinas terem tabelas inconsistentes durante o intervalo de tempo compreendido entre o começo de uma inserção e seu fim, o que poderia levar à violação da unicidade de algum nome. Obviamente, o impacto no desempenho do sistema imposto por essa solução é inaceitável, o que levou à adoção de uma consistência fraca na TNUGE.

Contudo, essa consistência fraca significa que é possível que um nome possua muitas entradas na TNUGE, sendo que no máximo uma delas (e possivelmente nenhuma) esteja correta. Essa circunstância é agravada pelo fato de um nome não ser retirado de nenhuma TNUGE do domínio quando ele deixa de existir. Assim, a TNUGE poderia começar a crescer indefinidamente. A fim de evitar isso, toda vez que certas condições são satisfeitas, o *daemon* percorre a lista de máquinas ativas do domínio e confirma com cada máquina todos os nomes que constam na TNUGE referentes àquela máquina.

A fim de saber o momento de iniciar uma coleta de lixo, o *daemon* mantém uma tabela com os *N* últimos tamanhos que a TNUGE assumiu imediatamente após as coletas de lixo. Ou seja: cada vez que é feita uma coleta de lixo, guarda-se o número de elementos da TNUGE nessa tabela e descarta-se o valor mais antigo. Assim, a tabela tem um his-

tórico de qual o tamanho da TNUGE após as últimas N coletas de lixo. Dentre os N valores da tabela, escolhe-se N_{max}, o maior deles, e multiplica-se esse valor por um número real R. Toda vez que o tamanho da TNUGE exceder o valor $N_{max}*R$, é iniciada uma coleta de lixo. Existe também um tempo máximo T_{max} que a tabela pode permanecer sem uma coleta de lixo. Toda vez que esse tempo é excedido sem ter sido feita nenhuma coleta, o *daemon* dispara uma. Os valores de N, R e T_{max} são configuráveis pelo administrador do sistema e têm por *default* $N = 4$, $R = 1.8$ e $T_{max} = 24$ horas.

4.4.8 Prevenção de *Deadlocks*

Se vários clientes requisitarem serviços simultaneamente a diversos *daemons*, eles poderão tentar comunicar-se mutuamente ao mesmo tempo. Imagine o seguinte cenário: *daemons A* e *B* enviam um *broadcast* no domínio no mesmo instante. O *daemon A* está aguardando que todas as máquinas lhe enviem um resposta. Em particular, *A* espera uma resposta de *B*. O mesmo acontece com *B*, que aguarda uma resposta de *A*. Caso a implementação do Servidor de Nomes não seja capaz de lidar com essa situação, um *daemon* ficará esperando a resposta do outro sem que nenhum deles nunca responda, fazendo com que entrem em *deadlock*.

A melhor maneira de evitar essa situação é fazendo com que os *daemons* do Servidor tenham mais de um fluxo de execução. Dessa forma, um fluxo pode, por exemplo, ficar bloqueado aguardando as respostas enquanto outro responde às requisições que lhe forem feitas.

Várias são as maneiras de fazer com que os *daemons* tenham mais de um fluxo de execução:

- O *daemon* pode ser composto por dois ou mais processos que compartilham memória e sincronizam o acesso a ela através de semáforos;
- O *daemon* pode se constituir de dois ou mais processos que se comunicam através de mensagens;
- Novos processos podem ser criados sob demanda. Quando precisasse fazer um *broadcast*, por exemplo, o *daemon* poderia criar um processo apenas para isso. Ele faria o *broadcast*, coletaria todas as respostas e terminaria;
- O *daemon* poderia ter mais de um fluxo de execução dentro do mesmo processo. Isso pode ser conseguido através de facilidades como *Multithreads* da Sun [MT93] ou equivalentes, que permitem a criação de várias *threads* dentro do mesmo processo.

Ao nosso ver, a solução usando *Multithreads* é a melhor, pois deixaria o programa mais eficiente e limpo. Essa solução tem outros problemas de ordem prática, entretanto: *Multithreads* só estão disponíveis no Solaris 2.x, não existindo no SunOS 4.x[1] e nem na

1. A biblioteca de Processos Leves (*Lightweight Processes*) [LWP90] existente no SunOS 4.x *não* é o mesmo que as *Multithreads* do Solaris e não poderia ser usada para resolver esse problema.

maioria das versões de UNIX de outros fabricantes, portanto seu uso comprometeria bastante a portabilidade do sistema. Além disso, essa facilidade não funciona[1] em conjunto com RPC (*Remote Procedure Call*) nem com *sockets*, sendo TLI (*Transport Level Interface*), portanto, o único mecanismo de comunicação em conjunto com o qual poderia ser usada. O DCE (*Distributed Computing Environmemt*) [DCE90] também provê facilidades semelhantes, mas essa ferramenta não está disponível para nós.

Excetuando *Multithreads*, a melhor solução é a de ter dois processos compartilhando memória, pois é mais eficiente e um pouco mais simples de implementar. Qualquer que seja a solução adotada, entretanto, ela sem dúvida deixa a implementação do Servidor de Nomes consideravelmente mais complicada do que se fosse ignorada a ocorrência de *deadlocks*.

1. "Por enquanto", segundo a Sun.

Capítulo 5

Portas de Comunicação

As Portas de Comunicação do OMNI são responsáveis por permitir a troca de mensagens entre processos distribuídos. Toda porta possui uma OMNIid, que a identifica unicamente no tempo e no espaço e pode ou não ter um nome simbólico registrado no Servidor de Nomes. Dois tipos de portas existem no sistema: Portas Conectáveis e Portas Não Conectáveis. A seguir explicaremos cada um desses tipos em detalhe.

5.1 Portas Conectáveis

Para que dois processos se comuniquem usando Portas Conectáveis, é necessário que cada um deles tenha uma porta e que seja estabelecida uma conexão entre elas. Não é necessário que o processo que estabeleceu a conexão seja um dos donos das portas. Uma vez conectadas, um dado escrito em uma porta pode ser lido na outra.

Portas conectáveis estendem o conceito de *pipes* para um ambiente distribuído. No UNIX, quando é executado na *shell* o comando

```
% ls | more
```

a *shell* cria um pipe e dois processos, redirecionando a saída padrão do primeiro para a entrada padrão do segundo. Ou seja: a *shell* conecta os processos `ls` e `more` sem que estes sejam notificados disso. Não há nada especial no código desses programas para gerenciar a conexão. Eles operam da mesma forma se estiveram lendo ou escrevendo no terminal ou num *pipe*.

Com Portas de Comunicação OMNI é possível fazer o mesmo com processos distribuídos, ou seja, um processo *A* pode conectar as portas de dois processos *B* e *C*. Isso não quer dizer que *B* não pode tomar por si próprio a iniciativa de se conectar a *C*.

Um processo pode possuir um número arbitrário de portas, estando apenas limitado pelo número máximo de descritores de arquivo que o processo é capaz de abrir.

5.1.1 Primitivas para a Manipulação de Portas Conectáveis

Abaixo descrevemos sucintamente os principais serviços oferecidos pelo módulo de Portas Conectáveis:

- `om_init`: inicializa o processo para que possa usar o sistema OMNI;
- `om_createConPort`: cria uma porta;
- `om_destroyConPort`: destrói uma porta;
- `om_connect`: conecta duas portas;
- `om_filePort`: associa uma porta a um nome de um arquivo;
- `om_read`: lê dados de uma porta;
- `om_write`: escreve dados numa porta;
- `om_disconnect`: encerra uma conexão;
- `om_destroy`: destrói uma porta;
- `om_waitConn`: bloqueia até que todas as portas (ou até que uma porta específica) do processo sejam conectadas.

Além dessas primitivas, que lidam especificamente com portas, as seguintes funções manipulam conectores especiais (o conceito de conectores especiais será explicado na seção 5.1.5):

- `om_createMbox`: cria um conector especial tipo *mailbox*;
- `om_createBcast`: cria um conector especial tipo *broadcast*;
- `om_destroyCon`: destrói um conector especial.

5.1.2 Uso de Portas Conectáveis

Ao começar a executar, um processo cria suas portas utilizando a primitiva `om_createConPort`. Ele pode então conectar suas portas às de outros processos ou esperar que alguém as conecte. Se ele tomar a iniciativa de conectá-las, deve usar a primitiva `om_connect`. Caso espere que outro processo as conecte, ele pode usar `om_waitConn` para esperar que as conexões sejam completadas ou simplesmente tentar ler ou escrever usando `om_read` ou `om_write` nas portas desconectadas. Uma operação dessas numa porta desconectada bloqueia até que alguém a conecte. Finda a comunicação, a porta pode ser desconectada com `om_disconnect` e, se não for mais ser usada, destruída com `om_destroy`.

5.1.3 Arquivos

É possível associar uma porta a um nome (*pathname*) de arquivo através da primitiva `om_filePort`. Essa categoria de porta é sempre unidirecional (só de escrita ou só de leitura). Quando uma porta desse tipo é conectada, o conteúdo do arquivo é enviado

pela conexão, caso a porta seja de escrita, ou os dados que chegam pela conexão são armazenados no arquivo, caso seja de leitura.

Note que não está sendo criada uma nova porta para o processo, uma vez que o arquivo é uma entidade *externa* a esse processo. A primitiva `om_filePort` apenas informa o sistema que o arquivo em questão deve ser encarado como uma porta de comunicação e, no futuro, poderá ser conectado a portas pertencentes a processos.

5.1.4 Atributos de uma Porta Conectável

Uma porta conectável é identificada por uma OMNIid e possui os seguintes atributos:

- Direção: a porta pode ser de entrada (leitura), saída (escrita) ou ambos;
- Semântica: este atributo diz respeito à estrutura dos dados que trafegam pela porta. As várias semânticas serão explicadas logo adiante;
- Nome (opcional): uma porta pode ter um nome simbólico (associado à sua OMNIid) cadastrado no Servidor de Nomes.

A seguir são detalhadas as possíveis semânticas de uma porta:

5.1.4.1 Semânticas

Os mecanismos comuns de comunicação entre processos do UNIX, como *pipes*, *sockets* e TLI com TCP/IP, não impõe fronteiras de mensagens sobre os dados que trafegam em uma conexão. No caso do OMNI, entretanto, existem situações onde é essencial saber onde acaba uma mensagem e começa a próxima (isto é particularmente necessário quando utilizam-se conectores especiais, explicados na seção 5.1.5). Assim sendo, o OMNI introduz o conceito de *semânticas* dos dados que trafegam por uma porta. A semântica impõe uma estrutura mínima sobre esses dados, permitindo assim derivar fronteiras entre as mensagens.

As possíveis semânticas são:

- *Stream*: os dados são uma seqüência de *bytes* sem estrutura. Ou seja, qualquer bloco contíguo de *bytes* da seqüência pode ser considerado uma mensagem;
- Bloco máximo: todos os *bytes* enviados entre o início e o término de uma conexão são considerados uma única mensagem;
- Tamanho fixo: cada mensagem tem sempre o mesmo tamanho fixo em *bytes*, parametrizável para cada porta;
- Caractere separador: as mensagens são separadas umas das outras por um caractere (*byte*) com valor específico. É possível especificar um conjunto de caracteres que determinam o fim da mensagem. Ou seja: quando algum dos caracteres do conjunto surgir na mensagem, considera-se que esta acabou;
- Tamanho variável: cada vez que é efetuada uma escrita numa porta (ou seja, cada vez que é ativada a primitiva `om_write`) é passado um argumento informando se a mensagem terminou ou não. Por exemplo: o usuário poderia ativar `om_write` três vezes, sendo que nas duas primeiras ele passa o argumento dizendo que a mensagem ainda não acabou e na última ele o passa avisando que a mensagem foi conclu-

ída. A mensagem seria então constituída pela concatenação dos dados enviados em cada uma das três ativações de `om_write`;

- Herdada: a semântica da porta permanece indefinida até que ela seja conectada a uma outra, quando então assume (herda) a semântica dessa porta. Se duas portas com semântica herdada são conectadas, ambas assumem semântica de *stream*.

5.1.5 Conectores Especiais

Um conceito novo implementado pelo OMNI (introduzido na *LegoShell*) é a idéia de conectores especiais. É possível no UNIX criarmos uma cadeia (*pipeline*) de processos interligados por *pipes*, onde a saída de cada um liga-se à entrada do seguinte. Além disso, pode-se fazer mais de um processo ler ou escrever num único *pipe*. O sistema permite tal arranjo, mas provê pouco ou nenhum suporte para delimitar as mensagens dentro do *pipe*. Os processos envolvidos é que seriam os responsáveis por implementar regras que evitassem essa mistura. Isso, aliado ao fato de que *pipes* UNIX são incapazes de interligar processos remotos, restringe bastante a utilização adequada desse recurso.

O OMNI implementa conectores especiais que procuram sanar os problemas apontados. Podemos ligar quantas portas quisermos (pelo menos uma) tanto à entrada quanto à saída de um deles conforme mostrado na figura 3. O sistema suporta dois tipos de conectores especiais:

- *Broadcast:* As portas de todos os processos C_i, $1 \le i \le m$ ligadas à saída do conector K recebem uma cópia de todos os dados enviados por cada uma das portas dos processos P_j, $1 \le j \le n$ ligadas à entrada de K;
- *Mailbox*: Somente uma porta pertencente a algum processo C_i, $1 \le i \le m$ ligada à saída de K recebe um dado que entrou por uma porta pertencente a algum processo P_j, $1 \le j \le n$ ligada à entrada de K. Quando um processo lê um dado de uma porta conectada ao *mailbox*, ele consome esse dado, impedindo que ele seja lido novamente em outra porta.

Com esses dois tipos de conectores, é possível generalizar conexões unidimensionais (*pipelines*) para conexões bidimensionais.

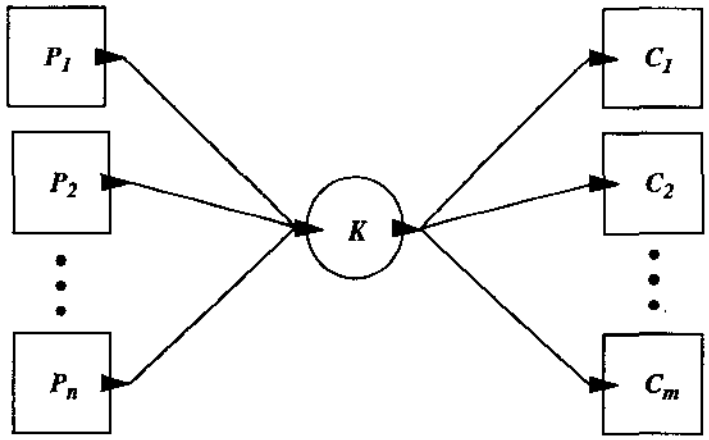
Se não houvesse uma semântica associada à cada porta liga à entrada do conector, este não conseguiria determinar as fronteiras entre as mensagens de cada processo e misturaria completamente os dados dentro de si, sendo incapaz de enviar dados corretos às portas ligadas à sua saída.

5.1.5.1 Ordenação Entre as Mensagens

No caso do conector de *broadcast*, todas as portas ligadas à saída do conector recebem todas as mensagens exatamente na mesma ordem. Garante-se que se uma mesma porta enviou duas mensagens numa certa ordem, esta ordem é preservada.

Já no caso de conectores *mailbox* garante-se apenas que, se uma porta A ligada à saída do conector receber duas mensagens enviadas por uma mesma porta B ligada à entrada do conector, essas mensagens terão chegado em A na mesma ordem relativa que foram enviadas por B.

FIGURA 3 Conector especial K interconectando n processos produtores $P_1, P_2, \ldots, P_n$ e m processos consumidores $C_1, C_2, \ldots, C_m$.



5.1.5.2 Conectores Especiais Locais

Até agora, não dissemos como os conectores de *broadcast* e *mailbox* podem ser implementados, mas veremos mais adiante que isso pode ser feito de mais de uma maneira. Na medida do possível, deve ser irrelevante para o usuário como os conectores estão de fato implementados. Entretanto, por razões puramente de facilidade de utilização pelo usuário e de eficiência, o OMNI provê um tipo particular de conector especial chamado de *conector local*. Este conector é sempre implementado por um processo local à máquina onde reside o processo que o criou e este fato é feito não-transparente ao usuário intencionalmente. A função do conector local é permitir que portas conectáveis de outros processos possam ter suas entradas/saídas facilmente redirecionadas para descritores de arquivo do processo local. Este recurso é particularmente útil quando se deseja executar um processo remotamente e fazer com que o resultado de sua execução seja mostrado na saída padrão local, ou quando se deseja disseminar dados da entrada padrão local para processos remotos.

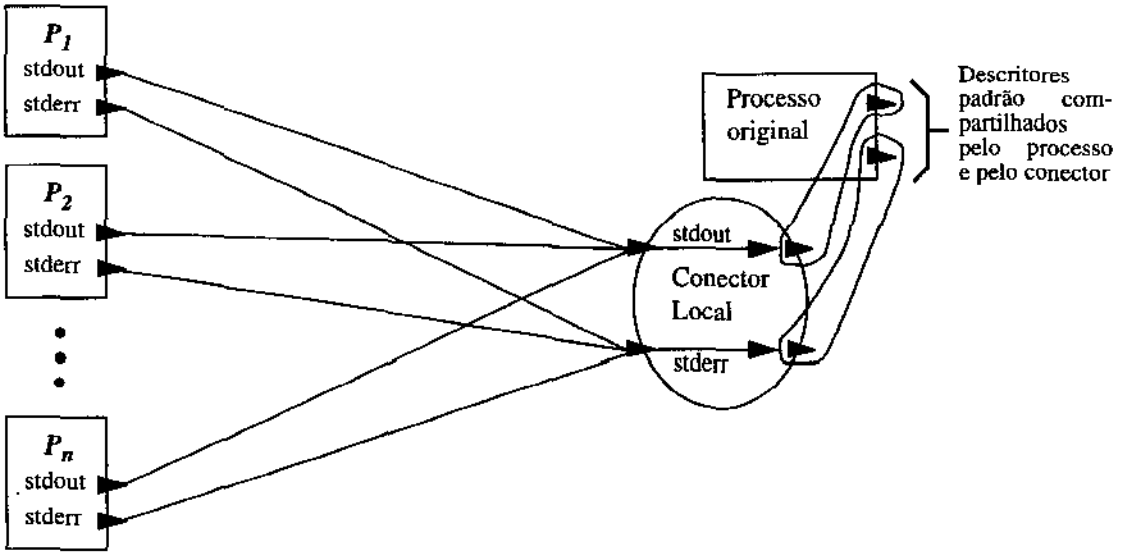
Um conector local é um processo criado por uma primitiva OMNI que se utiliza de uma chamada ao sistema `fork` e que, portanto, herda todos os descritores de arquivo abertos de seu processo pai. No momento da criação, é possível especificar quantas portas o conector irá possuir, quais delas são de entrada ou saída e associar cada uma delas a um descritor de arquivo. Portas de entrada escrevem no descritor tudo o que recebem, ao passo que portas de saída enviam o que lêem de seu descritor associado para uma (ou para todas) as portas ligadas à sua saída (conforme a semântica do conector local seja de *mailbox* ou *broadcast*).

Exemplo

Conforme veremos no capítulo 6, o OMNI permite a criação de processos em qualquer máquina da rede, sendo possível redirecionar seus descritores para portas de comunicação. Um processo poderia, por exemplo, criar uma série de outros em várias máquinas,

redirecionando as suas saídas padrão e de erro para portas de comunicação. Criaria também um conector local com duas portas de entrada, uma associada à saída padrão e outra à saída de erro do processo local. Por fim, conectaria as portas de todos os processos remotos associadas a suas saídas padrão à porta do conector local associada à saída padrão local. O mesmo seria feito com as saídas de erro. O efeito disso é que tudo que qualquer um dos processos remotos escrevessem em seus descritores padrão seria transmitido pela rede e escrito nos descritores padrão do processo local. Essa situação encontra-se ilustrada na figura 4

FIGURA 4 Conector local usado para redirecionar as saídas padrão e de erro de processos remotos para os descritores de seu processo criador.



5.2 Portas Não Conectáveis

Portas não conectáveis são uma maneira alternativa de enviar mensagens entre processos. No caso anterior de portas conectáveis, cada processo escreve e lê em suas próprias portas, estando as portas do leitor e do escritor conectadas entre si. Já no caso de portas não conectáveis essa conexão (obviamente) não existe, e cada processo escreve diretamente nas portas dos outros, que são sempre de entrada (não existem porta de saída).

A menos de questões de eficiência e desempenho, não existe motivo para termos mais um tipo de porta além das conectáveis. As portas não conectáveis sacrificam confiabilidade em troca de eficiência. Também sacrificada fica a possibilidade de um terceiro processo determinar o destino das mensagens, pois, no caso de portas conectáveis, um processo pode conectar as portas de dois outros sem que o usuário tenha que escrever nenhum código especial para isso nos programas donos das portas. Já no caso das não conectáveis, como os processos escrevem diretamente nas portas dos outros, o processo

remetente precisa saber a OMNIid da porta destino, não podendo esse *binding* ser feito automaticamente por um terceiro processo.

Uma porta conectável garante a entrega de todas as mensagens uma e somente uma vez cada, na ordem em que foram enviadas, provendo também fluxo de controle (se o escritor é mais rápido que o leitor, o primeiro bloqueia até que o segundo tenha consumido mais alguns dados, como se houvesse um *buffer* entre eles). Já as portas não conectáveis não garantem a entrega da mensagem e, caso a mensagem seja entregue, não se garantem que ela chegará apenas uma vez. Também não há garantia que duas mensagens enviadas em uma certa ordem relativa não cheguem na ordem inversa. Não há nenhum fluxo de controle, de forma que se o escritor for mais rápido que o leitor, algumas mensagens serão perdidas.

Muitas aplicações não necessitam da confiabilidade oferecida pelas portas conectáveis e não estão dispostas a pagar o custo envolvido em mantê-la. Imagine, por exemplo, uma aplicação que recebe mensagens em uma porta, onde essas mensagens são *frames* de uma animação que está sendo mostrada na tela em tempo real. Se um *frame* não está disponível no momento exato de ser exibido ele torna-se inútil, não havendo motivo para se perder tempo tentando retransmiti-lo.

Note que na proposta inicial do Sistema OMNI (conforme descrito em [DC92]), as portas não conectáveis seriam tão confiáveis quanto as conectáveis. Experiências com o protótipo do sistema mostraram que dessa forma os dois tipos de portas seriam redundantes e preferiu-se mudar a especificação das portas não conectáveis para permitir que elas oferecessem um grau de eficiência maior em troca da perda de confiabilidade.

5.2.1 Primitivas para a Manipulação de Portas Não Conectáveis

Os principais serviços oferecidos pelo módulo de Portas Não Conectáveis são:

- `om_init`: inicializa o processo para que possa utilizar o sistema OMNI;
- `om_createConLessPort`: cria uma porta;
- `om_destroyConLessPort`: destrói uma porta;
- `om_send`: envia uma mensagem a uma porta;
- `om_receive`: recebe uma mensagem em uma porta.

A primitiva `om_createConLessPort` cria uma porta não conectável. Diferentemente das conectáveis, só existem portas não conectáveis de entrada. Um processo pode enviar mensagens a qualquer porta, mas só pode receber mensagens em suas próprias portas. Para enviar, ele deve utilizar a primitiva `om_send`, que recebe como parâmetro a porta destino. Para receber, ele deve se valer de `om_receive`, passando como argumento a porta onde recebeu a mensagem.

Além dessas primitivas, que lidam especificamente com portas, as seguintes funções manipulam grupos de portas (o conceito de grupos de portas será explicado na seção 5.2.2):

- `om_createPortGroup`: cria um grupo de portas conectáveis;

- `om_insertPort`: insere uma porta em um grupo;
- `om_removePort`: retira uma porta de um grupo;
- `om_destroyPortGroup`: destrói um grupo de portas.

5.2.2 Grupos de Portas

Grupos de portas não conectáveis desempenham a mesma função de conectores especiais para portas conectáveis. Um grupo de portas é um conjunto de zero ou mais portas não conectáveis que pode ser referenciado exatamente da mesma maneira que uma única porta. Assim como no caso de conectores especiais, dois tipos de grupos de portas são suportados:

- *Broadcast*: uma mensagem enviada ao grupo é recebida por todas as portas pertencentes ao grupo;
- *Mailbox*: somente uma porta recebe uma mensagem enviada ao grupo.

Os grupos de portas do OMNI foram inspirados no mesmo conceito do Sistema Chorus [RAA90], e são bastante semelhantes aos por ele implementados, com exceção de que as portas do Chorus são confiáveis (isto é, garantem a entrega das mensagens).

Uma das vantagens deste sistema, segundo os projetistas do Chorus, é que ele permite a divisão de carga e criação de serviços persistentes. De posse da identificação do grupo, um cliente faz requisições de serviço como se as estivesse fazendo diretamente ao servidor. Entretanto, a requisição é encaminhada a um dos membros do grupo e atendida. Se o grupo tem muitos membros, isso permite que eles dividam o trabalho entre si e possibilita que o serviço como um todo continue funcionando mesmo se um dos servidores cair, ficando a falha desse servidor transparente ao usuário (a menos que o processamento da requisição já tivesse começado).

No OMNI, esse tipo de facilidade exige um esforço maior do programador, pelo fato das portas não conectáveis não serem confiáveis. Em níveis mais altos do Ambiente A_HAND, como no nível do programador Cm, por exemplo, esse tipo de facilidade estará disponível num grau de abstração ainda maior do que o do Sistema Chorus. Isso porque os usuários terão a possibilidade de executar ativações de métodos de objetos remotos, onde um "objeto" na verdade poderá estar encapsulando um grupo de outros objetos. Isso será feito de forma distribuída, tolerante a falhas e transparente.

5.3 Especificação Interna do Módulo de Portas

O módulo de portas é implementado por uma biblioteca de funções em C e é dividido em dois submódulos: o de portas conectáveis e o de portas não conectáveis. No caso do usuário atribuir um nome simbólico a uma porta, os submódulos fazem uso do Servidor de Nomes para cadastrá-la e assim deixar sua OMNIid disponível a potenciais usuários.

Explicaremos agora os algoritmos, estruturas de dados e protocolos utilizados para implementar tanto as portas conectáveis quanto as não conectáveis.

5.3.1 Estruturas de Dados Comuns aos Dois Submódulos

Para utilizar qualquer serviço do sistema OMNI, e em particular para criar portas de comunicação, todo processo primeiro precisa se inicializar com uma chamada à função `om_init`. Esta chamada cria uma Tabela de Portas (TP) com um número de entradas igual ao número máximo de descritores de arquivo abertos que o processo pode ter. Cada entrada da TP possui os seguintes campos:

1. Conectibilidade: diz se a porta é conectável ou não conectável;
2. Uqid: é a identificação única da porta (conforme explicado na seção 4.1), que a distingue de qualquer outra;
3. *Capability*: número aleatório usado para aumentar a segurança do sistema;
4. Descritor de arquivo: descritor na tabela de arquivos abertos associado à porta;
5. Direção: diz se a porta é de entrada, saída ou ambos;
6. Semântica: indica qual a semântica associada à porta (veja seção 5.1.4.1);
7. Semântica do parceiro: no caso de estar estabelecida uma conexão, indica qual é a semântica da porta à qual está conectada;
8. Informação adicional sobre semântica: quando a semântica é de "tamanho fixo" é guardado aqui qual o tamanho de cada mensagem. Se ela é de "caractere separador" são guardados os caracteres usados para separar uma mensagem da outra. Dependendo do caso, esta informação adicional pode se referir à semântica da própria porta ou à da porta à qual ela está conectada;
9. Estado: Indica se no momento a porta está conectada ou não.

Os campos de 5 a 9 só têm significado no caso da porta ser conectável.

Além da TP, é criada também uma lista ligada (inicialmente vazia) contendo todos os pares (nome, uqid) que o processo obteve através de consultas ao Servidor de Nomes. Chamaremos essa lista de LNIU (Lista de Nomes e Identificações Únicas). Conforme veremos logo mais, esta lista permite que sejam removidas entradas inconsistentes no *Cache* de Nomes Externos (CNE) do Servidor de Nomes.

5.3.2 Interação com o Servidor de Nomes

Existem basicamente três situações nas quais o Módulo de Portas pode interagir com o Servidor de Nomes:

- No momento da criação de uma porta, se o usuário atribui a ela um nome, o par (nome, OMNIid) da porta é cadastrado no Servidor de Nomes;
- Se um usuário utiliza a primitiva `om_getId` para obter a OMNIid de uma porta a partir do seu nome, é feita uma consulta ao Servidor para consegui-la. Feito isso, é também extraído o campo de identificação única (uqid, veja seção 4.1) da OMNIid recém adquirida e o par (nome, uqid) é inserido na LNIU;
- Se ocorrer algum erro na comunicação com uma porta (seja ela conectável ou não) que possa significar que a porta já não mais existe, os seguintes passos são executados:

- a primitiva OMNI que incorreu no erro extrai a uqid da porta a partir da sua OMNIid;
- acha-se a entrada da LNIU que corresponde a essa uqid;
- de posse do nome da porta, efetua-se um *purge* desse nome no *Cache* de Nomes Externos (CNE, veja seção 4.4.1) do Servidor de Nomes;
- remove-se essa entrada da LNIU.

Conforme havíamos dito na seção 4.4.5, a consistência do *Cache* de Nomes Externos (CNE) do Servidor de Nomes é deixada a cargo das camadas superiores do OMNI. É portanto aqui no Módulo de Portas (bem como no de Gerenciamento de Processos) que é feita sua consistência.

5.3.3 Módulo de Portas Conectáveis

Mostraremos agora as estruturas e protocolos específicos do módulo de portas conectáveis.

5.3.3.1 Estruturas de dados

Além da TP, o submódulo de portas conectáveis utiliza as seguintes estruturas:

- Endereço de Controle (EC): no momento da inicialização, é atribuído ao processo um endereço TCP/IP (que pode estar associado a um *endpoint* TLI ou a um *socket*) que será utilizado para permitir que outros processos se comuniquem com ele para estabelecer conexões entre suas portas. O Endereço de Controle é global para o processo;
- Dentro da OMNIid de cada porta, o campo "tipo da entidade" (veja seção 4.1) assume o valor correspondente a "porta conectável" e o campo "informação específica" possui três subcampos:
  - Endereço de Controle (EC): este é o endereço TCP/IP associado ao processo, usado para estabelecer conexões com ele;
  - direção (veja seção 5.1.4): diz se a porta é de entrada, saída ou ambos;
  - índice na TP: índice da entrada correspondente à porta na TP do processo que a possui.

No caso de portas associadas a arquivos (veja seção 5.1.3), o campo "tipo da entidade" tem valor "arquivo" e "informação específica" tem os campos:

- Nome (*pathname*) completo do arquivo;
- *Flags* com as quais o arquivo deve ser aberto (O_RDONLY, O_CREAT, O_TRUNC, etc. Veja [referência open(2V)]). A *flag* O_RDWR não é aceita, uma vez que a porta é sempre apenas de leitura (saída) ou escrita (entrada);
- Modo de proteção do arquivo (necessário somente no caso em que o arquivo ainda não existe e será criado);
- Semântica da porta;
- Informação adicional sobre semântica.

Note que uma porta associada a um arquivo não pertence ao processo e não tem uma entrada associada a ela na TP do processo.

5.3.3.2 Criação de uma porta

Para criar uma porta, o usuário deve utilizar a primitiva `om_createConPort`, especificando para ela todas as características da porta, que são:

- Direção;
- Semântica. Algumas semânticas exigem informação adicional:
  - para a semântica de "tamanho fixo" é também necessário especificar o tamanho;
  - para a de "caractere separador", é preciso fornecer os caracteres separadores;
- Nome. Este é um atributo opcional da porta. Caso seja especificado, ela será registrada no Servidor de Nomes. Nesse caso é também necessário fornecer os seguintes atributos do nome (veja seção 4.3.3):
  - unicidade;
  - visibilidade;
  - permissões de acesso.

A função irá atualizar a TP do processo com essas informações e criar uma OMNlid para a porta, que é devolvida ao usuário. No caso de ter sido atribuído um nome à porta, o par (nome, OMNlid) da porta é registrado no Servidor de Nomes.

5.3.3.3 Protocolo de estabelecimento de conexão

Existem dois casos (e dois protocolos) para o estabelecimento de conexão entre portas, que são ambos implementados pela função `om_connect` (do lado de quem está iniciando a conexão) e pelas funções `om_waitConn`, `om_read` ou `om_write`[1] (do lado de quem não toma a iniciativa de conexão). A situação mais simples ocorre quando um processo deseja conectar uma de suas portas a uma porta de outro processo. Nesse caso o protocolo é o seguinte:

Seja A o processo que deseja conectar sua porta P_a à porta P_b de B. O processo A deve conhecer as OMNlids de ambas as portas. Os passos executados pelo protocolo são:

- Usando o campo EC (Endereço de Controle) da OMNlid de B, A estabelece uma conexão com B;
- O processo A envia para B uma mensagem com o seguinte conteúdo:
  - código de controle informando que a mensagem é um pedido de conexão entre as portas de A (que é quem está iniciando a conexão) e B;
  - o índice de P_b na TP de B (extraído da OMNlid de P_b);
  - a identificação única (uqid) de P_b (extraída da OMNlid de P_b);
  - a *capability* de P_b (extraída da OMNlid de P_b);

1. Lembre-se que chamar `om_read` ou `om_write` para uma porta que ainda não está conectada bloqueia a função até que a porta seja conectada.

  - a semântica de P_a, junto com a informação adicional sobre semântica, caso haja alguma (extraídas da TP de A).
- O processo B verifica a uqid e a *capability* enviadas por A e só aceita a conexão se forem idênticas às que tem armazenadas em sua TP. Caso resolva estabelecer a conexão, B atualiza sua TP e responde com uma mensagem que tem o seguinte conteúdo:
  - código de controle informando que a conexão foi aceita;
  - semântica de P_b, junto com qualquer informação adicional sobre semântica, caso haja alguma (extraída da TP de B);
- O processo A atualiza a entrada da TP referente a P_a;
- A partir desse momento, tanto A quanto B passam a encarar essa conexão como sendo a conexão entre as portas P_a e P_b.

Note que a checagem que B efetua sobre a uqid enviada por A serve para ter certeza que ambos estão de fato referenciando a mesma porta. Já a checagem da *capability* tem como finalidade aumentar o grau de segurança do sistema pois, uma vez que ela é um número aleatório, somente aquele que realmente possuir a OMNIid da porta é que deverá ser capaz de usá-la, pois é improvável que uma *capability* escolhida ao acaso coincida com a certa. É claro que isso é um recurso simples e barato usado apenas para *dificultar* a penetração de intrusos no sistema, mas claramente insuficiente para torná-lo de fato seguro. Alguém capaz de observar as mensagens que trafegam pela rede, por exemplo, poderia examinar o conteúdo de uma mensagem e descobrir a *capability* de uma porta. A partir daí ele poderia usar a porta sem problemas. Somente utilizando criptografia seria possível elaborar um esquema com alto grau de segurança.

O segundo caso de estabelecimento de conexão é aquele em que um processo conecta as portas de dois outros. Nesse caso o protocolo é o seguinte:

Seja C um processo que deseja conectar a porta P_a pertencente ao processo A à porta P_b pertencente a B (C conhece as OMNIids de P_a e P_b). Os passos necessários para tanto são:

- O processo C inspeciona as OMNIids de P_a e P_b e encontra aquela cujo EC (Endereço de Controle) do processo é menor[1]. Sem perda de generalidade, iremos supor que o EC de A é menor. Esta precaução tem por finalidade evitar a ocorrência de *deadlocks* no caso de dois processos tentarem simultaneamente conectar P_a e P_b;
- O processo C estabelece uma conexão com A e envia para ele uma mensagem cujo conteúdo é:
  - código de controle informando que C deseja conectar uma porta de A à uma porta de um terceiro processo;

1. Para comparar dois endereços TCP, transforma-se o endereço Internet da máquina em um inteiro longo sem sinal e o número do *port* em um inteiro curto sem sinal. O endereço menor é o que tem menor endereço de máquina. Caso estes sejam iguais, toma-se o que tem menor número de *port*.

  - uqid, *capability* e índice na TP de P_a (extraídos da OMNlid de P_a);
  - EC de B (extraído da OMNlid de P_b);
  - uqid, *capability* e índice na TP de P_b (extraídos da OMNlid de P_b);
- O processo A checa se a uqid e a *capability* de P_a que recebeu são válidas. Se forem, procede exatamente como se tivesse ele mesmo tomado a iniciativa de conectar sua porta P_a a P_b e executa os passos descritos no caso anterior;
- Uma vez conectadas as portas, A simplesmente encerra a conexão com C (caso tivesse ocorrido algum erro na tentativa de conexão com P_b, A teria enviado uma mensagem com um código de erro a C antes de terminar a conexão).

Temos também o caso em que uma porta é conectada a um arquivo, ou melhor, à porta associada ao arquivo: nesse caso, a informação sobre o arquivo, que consta da OMNlid de sua porta, é enviada ao processo cuja porta será a ele conectada. O processo, por sua vez, encarrega-se de abrir o arquivo corretamente e atualizar sua TP de acordo.

5.3.3.4 Troca de mensagens

Para enviar uma mensagem, o usuário deve usar a primitiva `om_write` na sua porta de saída e para receber um mensagem, deve se valer de `om_read` na sua porta de entrada (uma porta pode ser de entrada e saída simultaneamente).

A função `om_read` deve interpretar os dados que recebe e dividi-los em mensagens, que serão retornadas como unidades lógicas independentes para o usuário. A divisão em mensagens deve ser feita com base na semântica da porta de entrada e possivelmente na semântica da porta de saída também.

No momento da conexão, os processos trocaram informações sobre suas respectivas semânticas e guardaram estas informações em suas TP. Agora, para que a divisão de mensagens seja possível, é necessário que os processos se coordenem com base nessas informações e enviem as mensagens corretamente:

- Se a semântica da porta de entrada for *stream*, bloco máximo, tamanho fixo ou caractere separador, a porta de saída deve enviar apenas o conteúdo da mensagem do usuário. O leitor desprezará a semântica da porta de saída e interpretará os dados somente segundo sua própria semântica;
- Se a semântica da porta de entrada for tamanho variável e a da porta de saída for *stream*, bloco máximo, tamanho fixo ou caractere separador, apenas o conteúdo da mensagem do usuário deve ser transmitido e o leitor dividirá as mensagens segundo a semântica da porta de saída;
- Se ambas as semânticas forem tamanho variável, então é necessário transmitir o tamanho de cada mensagem antes da mensagem propriamente dita. Isso é necessário porque o TCP não provê fronteiras entre mensagens;
- Se alguma das portas (ou ambas) tiver semântica herdada, sua semântica fica definida no momento da conexão e caímos num dos casos anteriores.

5.3.3.5 Desconexão

Um processo pode desconectar uma de suas portas através de `om_disconnect`. Não é possível desconectar a porta de outro processo. Após ser desconectada, a porta pode ser destruida usando-se `om_destroy`. Se uma porta conectada é destruida, a conexão é abortada, com possível perda de dados.

5.3.3.6 Conectores especiais

Conforme descrito na seção 5.1.5, existem dois tipos de conectores especiais: *broadcast* e *mailbox*. Duas abordagens são possíveis para implementá-los: a abordagem centralizada, onde um processo é responsável por coletar as mensagens e distribuí-las corretamente para cada porta destino; e a abordagem distribuída, onde desaparece a figura do processo centralizador. Nesta última, uma mensagem enviada a um conector de *broadcast* significa o envio de um *multicast* confiável com ordenação total na rede. Já no caso da mensagem enviada ao *mailbox*, significa decidir qual dentre as portas destino é a mais apta a receber uma mensagem e enviá-la somente para aquela porta.

Como pode-se perceber, a implementação dos conectores de forma distribuída é muito mais complexa do que a centralizada. Sendo assim decidimos, numa primeira abordagem, optar pela solução mais simples, centralizada. Já está claro, entretanto, que no futuro precisaremos de facilidades de *multicast* confiável com variados graus de ordenação.

Um conector especial é, portanto, um processo que possui duas porta pré-definidas: uma de entrada e outra de saída, ambas com semântica herdada. Essas portas têm uma particularidade que as diferencia das portas dos processos (criadas com `om_createConPort`): podemos conectar quantas portas quisermos a cada uma das portas de um conector, e isso equivale a criarmos uma nova porta cada vez que uma conexão é efetuada.

Cada uma dessas novas portas, onde é realmente estabelecida a conexão, herda a semântica da porta à qual ela foi conectada. Assim, o conector especial é capaz de dividir os dados que recebe nas portas de entrada em mensagens definidas e reenviar cada uma delas corretamente pelas suas portas de saída.

No caso do conector de *broadcast*, cada mensagem que chega por uma porta de entrada é replicada em todas as portas de saída. Já no conector de *mailbox*, a mensagem é reenviada a apenas uma das portas de saída. Para decidir qual, o conector escolhe, dentre as portas em que é possível escrever naquele momento sem que o processo bloqueie (caso exista mais de uma), aquela onde menos *bytes* foram enviados até agora.

5.3.3.7 Conectores locais

Um conector local é essencialmente um conector comum de *broadcast* ou *mailbox*, necessariamente implementado sob a forma de um processo, com pequenas modificações para suportar múltiplas portas (ao invés de apenas um par) e para utilizar-se dos descritores de arquivos herdados de seu pai. Conforme dito na seção 5.1.5.2, este conector precisa ser implementado por um processo para que ele possa ser criado por seu pai usando a chamada ao sistema `fork` e assim herdar seus descritores de arquivo, nos quais irá escrever/ler os dados que chegam/sairão por suas portas.

5.3.4 Portas Não Conectáveis

Mostraremos agora as estruturas e protocolos usados para implementar as portas não conectáveis.

5.3.4.1 Estruturas de dados

Além da TP, o submódulo de portas não conectáveis utiliza a identificação OMNI da porta. Dentro da OMNIid, o campo "tipo da entidade" (veja seção 4.1) assume o valor correspondente a "porta não conectável" e o campo "informação específica" possui dois subcampos:

- Endereço da porta: este é o endereço UDP associado àquela porta, para onde serão enviadas as mensagens;
- Índice na TP: índice da entrada correspondente à porta na TP do processo que a possui.

5.3.4.2 Criação de uma porta

O usuário cria uma porta não conectável usando a primitiva `om_createConLessPort`. Nesse momento pode opcionalmente ser atribuído um nome à porta, sendo que nesse caso é necessário especificar os atributos de visibilidade, unicidade e as permissões de acesso do nome

A função irá atualizar a TP do processo com essas informações e criar uma OMNIid para a porta, que é devolvida ao usuário. No caso de ter sido atribuído um nome à porta, o par (nome, OMNIid) é registrado no Servidor de Nomes.

5.3.4.3 Envio e recebimento de mensagens

Para enviar uma mensagem, o usuário utiliza-se da primitiva `om_send`, que recebe como parâmetros:

- A OMNIid da porta destino;
- A mensagem[1];
- O tamanho da mensagem em *bytes*.

A primitiva `om_send` extrai o endereço UDP da porta do campo de "informação específica" da OMNIid e envia para esse endereço um datagrama contendo:

- Índice da porta na TP;
- Uqid da porta;
- *Capability* da porta;
- A mensagem do usuário.

O processo destino recebe a mensagem usando a primitiva `om_receive`, que checa se a uqid e a *capability* da porta são idênticas às que se encontram na entrada da TP cujo

1. Mais precisamente, um apontador para o primeiro *byte* da mensagem.

índice foi fornecido. Em caso afirmativo, a mensagem é retornada ao usuário, caso contrário ela é descartada.

5.3.4.4 Grupos de portas

Conforme dito anteriormente, grupos de portas não conectáveis são equivalentes a conectores especiais para portas conectáveis. Assim sendo, as mesmas razões que levaram a optar por uma solução centralizada para os conectores se aplica aos grupos (veja seção 5.3.3.6).

Um grupo de portas é implementado por um processo possuidor de uma porta, que é criado através da primitiva `om_createPortGroup`. Quando uma mensagem chega nessa porta, o processo a reenvia para todas as portas membros do grupo (se o tipo do grupo for *broadcast*) ou apenas para uma delas (se o tipo for *mailbox*). Esse processo também aceita requisições para incluir uma nova porta no grupo (que são enviadas através de `om_insertPort`) ou para remover uma porta a ele pertencente (utilizando `om_removePort`).

Capítulo 6

Gerenciamento de Processos

O Gerenciador de Processos é o módulo do sistema OMNI responsável pela criação de processos remotos e pela troca de sinais entre eles. A pedido do usuário, ele pode também requisitar ao módulo de Portas que um ou mais descritores de arquivo de um processo sejam redirecionados para portas de comunicação. É possível também pedir que seja atribuído um nome simbólico a um processo, que é então registrado no Servidor de Nomes.

6.1 Descrição Funcional do Gerenciador de Processos

O Gerenciador é um programa distribuído constituído por um conjunto de *daemons* que executam em todas as máquinas da rede. cada *daemon* é responsável por criar e gerenciar os processos que residem em sua máquina. Os principais serviços que o gerenciador oferece são:

- `om_forkExec` e `om_createProc`: ambas as primitivas tem a mesma funcionalidade, que é a de criar um processo numa máquina remota. Elas apenas diferem na interface oferecida ao usuário. Enquanto `om_forkExec` tem número fixo de parâmetros e recebe com argumento uma estrutura que descreve o processo, `om_createProc` tem número variável de parâmetros e recebe como argumentos pares (atributo, valor) que especificam o processo;
- `om_initProc`, `om_copyProc`, `om_setProc`, `om_getProc`: manipulam a Estrutura de Descrição de Processo (EDP) a ser passada a `om_forkExec`:
  - `om_initProc`: inicializa uma EDP com os dados do processo corrente obtidos junto ao UNIX;
  - `om_copyProc`: copia uma EDP em outra;
  - `om_setProc`: atribui um valor a um campo de uma EDP. Os campos da EDP são os atributos do processo descritos na seção 6.1.1.2, página 54;

  - om_getProc: obtém o valor corrente de um campo de uma EDP;
- om_wait: espera pela morte de um processo;
- om_signal: especifica uma função tratadora de sinais para um sinal;
- om_kill: envia um sinal a um processo;
- om_killpg: envia um sinal a um grupo de processos;
- om_getpid: obtém a OMNIid do processo corrente;
- om_getppid: obtém a OMNIid do processo pai;
- om_getpgrp: obtém a OMNIid do grupo de processos ao qual pertence o processo corrente;
- om_setpgrp: muda o grupo ao qual pertence o processo corrente;
- om_setsid: cria uma nova sessão OMNI;
- om_enableSighup: habilita o envio do sinal SIGHUP a todos os membros de uma sessão por ocasião da morte do líder.

O Gerenciador de Processos do OMNI procura dar ao usuário a ilusão de estar trabalhando em uma só máquina, apesar de estar criando processos em toda a rede. Uma vez que o processo tenha sido criado, em momento algum é necessário explicitar sua localização. Na medida do possível, o Gerenciador oferece os mesmos serviços e abstrações existentes no UNIX. Assim é possível, por exemplo, enviar um sinal a um processo distribuído usando a primitiva om_kill, que recebe como parâmetros a OMNIid do processo e o número do sinal. Isso é muito semelhante à chamada de sistema kill do UNIX, onde são passados a identificação do processo (válida apenas na máquina local) e o número do sinal.

Abstrações como hierarquia entre processos (processo pai cria o processo filho e pode esperar pela sua morte), grupos de processos e sessões são também oferecidos de forma parecida ao UNIX.

6.1.1 Criação de Processos

Explicaremos agora como se dá a criação de processos no UNIX e em seguida, como ela é feita no OMNI, justificando as diferenças entre os dois sistemas.

6.1.1.1 Criação de Processos no UNIX

No UNIX, um processo é criado através da chamada ao sistema (*system call*) fork, que cria uma exata duplicata do processo que a chamou. No processo pai (processo criador), fork retorna a identificação do processo filho, ao passo que, no processo filho (processo criado), fork retorna zero. Isso permite que o código de ambos os processos, apesar de idêntico, tenha como distinguir se está executando o processo pai ou o filho.

Pai e filho possuem o mesmo código. A área de código é sempre de leitura apenas, o que permite que apenas uma cópia do código esteja fisicamente presente na memória e possa ser usada por ambos os processos. Na maioria das implementações de UNIX, as páginas de memória do segmento de dados são também compartilhadas, até o momento

em que um dos processos altere uma delas, quando então são criadas cópias distintas dessa página para cada processo.

Ambos os processos compartilham também seus descritores de arquivos abertos. Isso significa que, se o filho escrever em sua saída padrão, estará escrevendo no mesmo "arquivo" (que pode ser, na verdade, um *pipe*, um terminal, um dispositivo, etc) ao qual está direcionada a saída padrão do pai. O filho tem a opção de redirecionar seus descritores de arquivo como quiser, podendo assim abrir descritores distintos dos usados por seu pai.

Finalmente, é comum que o filho, após ter feito redirecionamentos e outros ajustes que julgar necessário, faça uma chamada ao sistema `exec`, que irá descartar totalmente tanto o código quanto a área de dados do processo antigo, substituindo-os pelo código e dados presentes no arquivo binário executável que contém o programa a ser posto em execução. Os descritores de arquivos abertos continuam válidos após a chamada `exec`.

Vejamos um exemplo concreto: analisemos o que acontece quando a *shell* (interpretador de comandos) executa a linha

```
% ls | more
```

- A *shell* inicialmente cria o *pipe* que irá ligar a saída padrão dos dois processos. Ao cria-lo, faz com que sejam abertos dois descritores de arquivo: um para a entrada e outro para a saída do *pipe*;
- A *shell* cria dois processos, P_1 e P_2, através de duas chamadas `fork`. Ambos os processos são idênticos à *shell* e herdam os descritores de arquivo referentes ao *pipe*;
- A *shell* original não precisa mais de nenhum dos dois descritores do *pipe*, por isso fecha ambos;
- A *shell* P_1 redireciona sua saída padrão para a entrada do *pipe*. Ela também fecha o descritor referente à saída do *pipe*, que não irá utilizar;
- A *shell* P_2 redireciona sua entrada padrão para a saída do *pipe*, fechando também a entrada do *pipe*, que não irá utilizar;
- P_1 chama `exec` e se transforma no programa `ls`, cuja saída padrão está redirecionada para o *pipe*;
- P_2 chama `exec` e "reencarna" no programa `more`, cuja entrada está redirecionada para o *pipe*.

Assim sendo, o processo `ls` irá enviar seus dados para o processo `more`, ao invés de fazê-lo para o terminal. O mais importante desse mecanismo é que nem o programa `ls` nem o `more` são os responsáveis pelo redirecionamento. Como foi um terceiro processo (no caso, a *shell*) o responsável, o código desses programas não tem que preocupar com essas situações nem possuir rotinas especiais para casos de redirecionamento, o que torna os programas muito mais limpos e modulares.

6.1.1.2 Criação de Processos no Sistema OMNI

O Sistema OMNI se propõe a operar sobre uma rede heterogênea de computadores com sistema UNIX, e a não alterar o núcleo do sistema operacional a fim de preservar a portabilidade. Dentro dessas premissas, é impossível implementar uma chamada equivalente a `fork` para um sistema distribuído. Os principais motivos para isso são:

- Uma vez que pai e filho podem estar executando em máquinas distintas e heterogêneas entre si, é impossível compartilhar o código. Isso porque as linguagens de máquina de cada computador podem ser diferentes, o que inviabiliza a solução de enviar uma cópia do segmento de código do processo de uma máquina para a outra;
- Não é possível também enviar os segmentos de dados de uma máquina para a outra, pois a maneira como os dados são representados em uma máquina pode não fazer sentido na arquitetura da outra. Poderiam surgir, por exemplo, problemas de tamanho dos dados, representação usada para os inteiros, alinhamento de variáveis na memória, etc;
- Não há como fazer com que os descritores de arquivo sejam compartilhados entre pai e filho. Isso seria bastante complicado de se fazer mesmo se tivéssemos a possibilidade de alterar o núcleo do UNIX. Sem esse recurso, torna-se realmente inviável.

Alguns desses motivos explicam também porque não se popularizou o recurso (a primeira vista muito interessante) de migração de processos, implementado por alguns sistemas operacionais como Sprite [DO87] ou V [TLC85]. Tais sistemas permitem que um processo em uma máquina seja transferido para outra sem interromper sua execução e de forma totalmente transparente ao usuário. Entretanto isso só funciona em ambientes homogêneos, o que é frontalmente contra a filosofia de sistemas abertos do UNIX. Note que implementar um `fork` remoto é essencialmente o mesmo que efetuar um `fork` local e em seguida migrar o processo para outra máquina.

A solução adotada pelo OMNI foi a de oferecer um serviço equivalente a um `fork`, seguido de quaisquer redireções e ajustes que o usuário deseje fazer, seguido de um `exec`. Dessa maneira, o que de fato ocorre é que o OMNI requisita a um *daemon* remoto que efetue o `fork`, as redireções/ajustes e `execute` o programa requisitado pelo usuário.

O que é possível especificar na criação do processo remoto

O sistema OMNI provê uma estrutura de dados chamada "descritor de processo" que especifica um processo a ser criado. Tal descritor vem com um valor *default* para cada atributo do processo, sendo esses valores (em sua maioria) herdados do processo corrente. Através das primitivas do sistema, é possível mudar qualquer dos *defaults* para um valor desejado. Quando o usuário tiver ajustado todos os valores conforme sua preferência, pode então mandar o sistema efetivamente criar o processo.

Os artibutos especificáveis para um processo são:

- Máquina da rede onde será executado;
- Nome do arquivo executável;
- Lista de argumentos do processo;

- Ambiente de execução (definição das variáveis de ambiente);
- Diretório corrente;
- Diretório raiz;
- Cada descritor de arquivo do processo pode ser redirecionado para um arquivo, para uma porta de comunicação ou simplesmente fechado. Caso seja redirecionado a uma porta, pode-se opcionalmente requisitar que a porta seja registrada no servidor de nomes;
- Identificação do usuário (uid) ou nome do usuário. Caso o usuário especificado seja diferente do local, os mesmos critérios usados por `rsh` (*remote shell*, veja `rsh(1C)` em [SRM90]) são usados para decidir se o usuário tem autorização de criar o processo;
- Prioridade de execução (*nice value*);
- Máscara de sinais ignorados;
- Máscara de sinais bloqueados;
- Máscara de modo de proteção dos arquivos (`umask`);
- Grupo OMNI do processo (é possível apenas especificar ele se herda o grupo do pai ou se inicia um novo grupo);
- Sessão OMNI do processo (é possível apenas especificar se permanece na sessão do pai ou se inicia uma nova);
- *Trace flag;*
- Todos os limites dos recursos (*resource limits*) que a versão do UNIX onde o processo irá executar permitir ajustar, tais como: tamanho máximo da tabela de descritores de arquivos abertos, tempo máximo de CPU, tamanho do maior arquivo que o processo pode criar, tamanho máximo do segmento de dados do processo, tamanho máximo da pilha, tamanho máximo do arquivo *core*, tamanho máximo do conjunto residente do processo, etc.

A primitiva de criação de processos retorna a OMNIid do processo recém criado. Caso algum descritor de arquivo tenha sido redirecionado para alguma porta de comunicação, uma lista com todas as OMNIids das portas (cada uma associada ao respectivo descritor de arquivo) é também retornada. Essas portas se encontrarão no estado desconectado e o processo ficará aguardando que todas elas sejam conectadas antes de efetuar o `exec` do novo código do processo. O uso mais comum é que o próprio processo pai conecte as portas do filho.

Assim como o UNIX, o OMNI mantém uma hierarquia entre os processos, sendo que o processo criador é o pai (*parent*) do processo filho (*child*), podendo esperar pela sua morte através da primitiva `om_wait` ou ser notificado desse evento assincronamente através do recebimento do sinal OM_SIGCHLD (nome usado pelo OMNI para o sinal SIGUSR2).

6.1.2 Envio de sinais

No UNIX, um sinal é similar a uma interrupção de *hardware*. É um evento geralmente assíncrono enviado ao processo e, quando seu recebimento não está bloqueado, pode fazer com que uma das seguintes ações seja tomada:

- O sinal é ignorado;
- O sinal termina o processo. Nesse caso, duas possibilidades existem:
  - É gerado um arquivo *core* contendo a imagem da memória virtual do processo quando o sinal ocorreu;
  - Não é gerado *core*;
- O fluxo de execução do processo é desviado para uma função previamente registrada que irá tratar o sinal.

Várias são as situações que podem causar a geração de sinais. Algumas delas ocorrem devido a ações internas efetuadas pelo próprio processo, como um acesso a uma posição ilegal de memória (SIGSEGV) ou uma divisão por zero (SIGFPE). Outras são causadas por eventos externos ao processo, como um CTRL-C teclado pelo usuário (SIGINT), a morte de um processo filho (SIGCHLD), ou o envio de um sinal por outro processo explicitamente através da chamada ao sistema `kill`. De especial interesse para o OMNI são os sinais passíveis de ser gerados externamente, pois esses são os que podem ser causados por processos distribuídos.

O OMNI procura estender o conceito de sinais do UNIX para um ambiente distribuído. Assim sendo, ele provê uma função `om_kill`, equivalente à chamada `kill` do UNIX, que envia um sinal a um processo dado o número do sinal e a identificação do processo. A única diferença é que no OMNI a identificação é a OMNIid.

Na seção 6.1.2.1 veremos que sinais podem também ser enviados a grupos de processos e que, assim como no UNIX, a morte de um processo líder de sessão causa o envio de um sinal para os processos membros da sessão. Na seção 6.1.3 veremos também que a morte de um processo filho pode causar o recebimento de um sinal no pai, da mesma maneira que no UNIX.

6.1.2.1 Grupos de Processos e Sessões

No UNIX existem os conceitos de sessão e grupo de processos. Uma sessão está tipicamente (mas não necessariamente) associada a um terminal de controle, sendo que é iniciada uma nova sessão quando o usuário digita seu nome e sua senha no *prompt* de *login*. Em ambientes de janelas, cada janela associada a um pseudo-terminal (onde normalmente encontra-se rodando uma *shell*) está associada a uma sessão diferente. Uma nova sessão pode ser criada através da chamada de sistema `setsid` [referência]. Quando um processo inicia uma nova sessão, ele se torna o líder dessa sessão. Dentro de cada sessão, processos são ainda agregados em grupos. Um grupo é um conjunto de processos compartilhando a mesma identificação de grupo.

Esses conceitos foram criados principalmente para permitir que a *shell* efetuasse um controle de tarefas (*job control*) sobre os processos, gerenciando o acesso ao terminal associado à sessão na medida que coloca grupos de processos para executar no "fundo"

(*background*) ou na "frente" (*foreground*) conforme o desejo do usuário (veja `termio(4)` em [SRM90]). No entanto, o conceito de terminal não tem razão de existir no OMNI. Se uma hipotética *shell* distribuída quisesse implementar controle de tarefas sobre os processos OMNI, ela deveria valer-se de Conectores Locais (veja seção 5.1.5.2) que seriam ligados aos descritores padrão dos processos remotos. O controle de tarefas seria então feito sobre os Conectores Locais, que não passam de processos executando localmente, usando os mecanismos convencionais do UNIX.

No entanto, controle de tarefas não é a única razão para agregarmos processos em sessões e grupos. Eles são úteis também para enviarmos sinais a um conjunto de processos, como veremos mais adiante.

O OMNI estende os parte dos conceitos de sessão e grupo de processos do UNIX para um ambiente distribuído. Ele descarta toda a funcionalidade referente a controle de terminais, concentrando-se apenas nos aspectos de troca de sinais.

Criando sessões e grupos

No OMNI, uma nova sessão é criada usando-se a primitiva `om_setsid`, que coloca o processo em um novo grupo e em uma nova sessão, onde ele é o único membro de ambos e líder da sessão. Note que essa função não efetua a chamada ao sistema `setsid` a fim de iniciar uma nova sessão UNIX. Caso seja isso o que o usuário deseje (constatamos que na maioria das vezes não é, pois o processo é desassociado de seu terminal controlador), ele deve chamar `setsid` explicitamente.

Uma vez que não existe *login* no sistema OMNI, quando um processo que não foi criado por nenhuma primitiva OMNI (como, por exemplo, aqueles criados a partir da linha de comando da *shell*) se inicializa usando `om_init`, o sistema automaticamente executa `om_setsid` para ele, colocando-o em um novo grupo e como líder de uma nova sessão. A partir daí, todo processo por ele criado através do OMNI herda sua sessão e grupo, a menos que seja explicitamente requisitado o contrário.[1]

Um novo grupo pode ser criado através de `om_setpgrp`, que coloca o processo chamador como seu único membro. Esta função também não altera o grupo do processo a nível de UNIX.

Trocando sinais

No UNIX, quando um líder de sessão termina, todos os membros recebem um sinal de *hangup* (SIGHUP), cuja ação *default* é terminar os processos. Cada processo pode, entretanto, decidir ignorar ou capturar esse sinal e continuar executando. No OMNI, todo líder de sessão pode pedir que seja enviado um sinal de *hangup* aos membros no momento de sua morte, mas isso não é feito por *default*. Caso seja requisitado, o OMNI passa a monitorar o processo e, quando ele termina (por qualquer motivo), envia auto-

1. Uma exceção para isso ocorre se o processo for criado em uma máquina que resida fora do domínio de *broadcast* da máquina corrente. Nesse caso, o processo é criado em uma nova sessão e grupo. Isso é feito para permitir que um sinal seja enviado a todos os membros do grupo ou sessão através de um *broadcast* no domínio. Futuras versões não deverão sofrer essa limitação.

maticamente o sinal aos outros. Optamos por não fazer o sinal ser enviado por *default* devido a dois motivos: constatamos que na maioria das vezes não era o que o usuário queria e por essa ser uma operação muito cara, uma vez que, da forma como foi implementada, envolve um *broadcast* na rede.

É possível também enviar explicitamente um sinal a um grupo de processos. No UNIX, a chamada que faz isso é `killpg`. No OMNI é `om_killpg`, totalmente análoga.

6.1.3 Espera pela morte de um processo

No UNIX, depois de criar um processo filho, o pai pode esperar pelo término de sua execução (morte) através da chamada ao sistema `wait`. O pai então bloqueia até que qualquer um de seus filhos morra,[1] recebendo como retorno a identificação do processo que terminou e mais algumas informações sobre seu estado nessa hora. O OMNI estende esse conceito, permitindo que o pai espere pelo término de um dos processos distribuídos por ele criados usando a primitiva `om_wait`, cuja funcionalidade é análoga à `wait` do UNIX. Assim como no UNIX, sucessivas chamadas a `om_wait` retornam as OMNIids dos processos na ordem em que estes vão morrendo.

No UNIX existe também uma maneira do processo ser avisado assincronamente da morte de seus filhos. Quando um filho termina, o sistema envia um sinal (SIGCHLD) a seu pai. Esse sinal é ignorado por *default*, mas o pai pode instalar uma função tratadora para capturá-lo. A única informação que o sinal transporta é a de que algum processo filho morreu, mas não qual deles nem com que estado de saída. Tipicamente essa função tratadora irá executar `wait` para obter a informação exata de qual filho terminou. Uma vez que é certo que pelo menos um filho tenha terminado, a chamada `wait` pode ser executada com a certeza de que irá retornar imediatamente.

No OMNI, também é possível ser avisado assincronamente da morte de algum filho através de um sinal. Optou-se pelo uso do sinal SIGUSR2, o qual rebatizamos de OM_SIGCHLD. Ele foi escolhido por ser um dos dois sinais guardados para utilização do usuário, e não ter nenhum significado especial para o UNIX. Assim sendo, caso deseje ser avisado assincronamente da morte de seus filhos, o usuário pode utilizar a primitiva `om_signal` para registrar um função tratadora para OM_SIGCHLD. Uma vez dentro da função, ele pode chamar `om_wait`, mantendo a analogia com o UNIX. Note que o sinal SIGURS2 (ou OM_SIGCHLD) só é de fato enviado ao processo se tiver sido instalado um tratador através de `om_signal`. Isso é diferente do que acontece com o SIGCHLD do UNIX, que é sempre enviado, mas cuja ação *default* é ser ignorado. Dessa maneira o OMNI permite que o usuário possa usar SIGUSR2 para outros fins caso não esteja interessado em receber avisos assíncronos da morte de seus filhos.

1. Caso o processo esteja sendo rastreado (i. e., esteja com a *trace flag* ligada) e mude de estado, `wait` também irá retornar acusando essa mudança (veja `ptrace(2)` em [SRM90]).

6.1.4 Controle de acesso

Uma facilidade de criação de processos remotos pode ser um séria brecha na segurança de qualquer rede, pois poderia permitir que intrusos disparassem processos sem a devida autorização. O OMNI adota a mesma política de controle de acesso do comando `rsh` (*remote shell*) (veja `rshd(8C)` em [SRM90]). Ou seja: se o usuário tiver permissão de criar uma *shell* remota na outra máquina sem ter que fornecer nenhuma senha, então ele pode criar um processo remoto naquela máquina usando o OMNI.

Note que utilizar a mesma política de controle de acesso não significa usar o mesmo mecanismo de autenticação de usuário remoto. Atualmente, o OMNI utiliza o mecanismo de autenticação equivalente ao provido pela biblioteca de RPC da Sun [NPG90] (na modalidade de "autenticação UNIX"), que é menos confiável que o usado pela `rsh`. Está prevista, entretanto, a inclusão de mecanismos baseados em criptografia de chave pública, como o RSA [Lc86], que o tornarão bem mais seguros do que uma `rsh`, que baseia-se no uso de números de porta privilegiados (endereços TCP/IP reservados apenas para o super-usuário).

6.1.5 Processos Comuns

O Sistema classifica os processos existentes em dois tipos: processos OMNI, que são aqueles que fazem uso de alguma primitiva OMNI (isto é, pelo menos de `om_init`), e processos comuns, que são aqueles que não usam nada do OMNI. Um processo comum normalmente não tem seu código ligado à biblioteca OMNI e pode ter sido implementado antes da criação deste, como é o caso dos comandos UNIX, como `ls`, `grep` ou `more`.

Apesar de não utilizar o OMNI diretamente, um processo comum pode ser criado por uma primitiva OMNI e pode ter seus descritores de arquivo redirecionados para portas de comunicação, o que permite que uma enorme gama de programas já existentes possam ser incluídos em computações distribuídas criadas usando o OMNI.

A capacidade de permitir a utilização de processos comuns é uma das principais diretivas de projeto do Sistema e, apesar de ter tornado sua elaboração mais complexa, é muito importante para torná-lo mais útil ao usuário.

6.2 Especificação Interna do Gerenciador de Processos

Além dos *daemons* que executam em cada máquina da rede, o Gerenciador é composto também por uma biblioteca de funções escrita em C ligada (*linked*) aos processos dos usuários. As estruturas de dados, protocolos e algoritmos usados para implementá-lo são explicados a seguir.

6.2.1 Estruturas de dados utilizadas pelo Gerenciador

As seguintes estruturas de dados são usadas para implementar o Gerenciador de processos:

6.2.1.1 Estruturas Comuns à Biblioteca e ao *Daemon*

As únicas estruturas de dados usadas tanto pela biblioteca quanto pelo *daemon* do Gerenciador são as OMNIids referentes às entidades que ele gerencia: processos, grupos de processos e sessões.

A OMNIid de um processo, além de seus campos padrão, possui as seguintes características:

- O campo "tipo da entidade" possui valor correspondente a "processo";
- O campo "informação específica" possui apenas um subcampo:
  - índice do processo na Tabela de Processos Correntes (TPC) do *daemon* que o gerencia.

As OMNIids dos grupos e sessões são parecidas:

- O campo "tipo da entidade" assume valor correspondente a "grupo de processos" ou "sessão", respectivamente;
- O campo "informação específica" é vazio.

6.2.1.2 Estruturas Específicas do *Daemon*

A principal estrutura de dados do *daemon* é a Tabela de Processos Correntes (TPC). Cada entrada dessa tabela descreve um processo correntemente sendo executado na máquina pela qual aquele *daemon* é responsável. Os principais campos de cada entrada da TPC são:

- Identificação UNIX do processo (pid);
- Identificação do usuário (uid) dono do processo;
- Identificação única do processo (uqid);
- *Capability* do processo;
- Identificação única (uqid) do pai do processo;
- Endereço da máquina onde se encontra o pai do processo;
- uqid do grupo do processo;
- *Capability* do grupo do processo;
- uqid da sessão do processo;
- *Capability* da sessão;
- Lista de portas do processo: caso um ou mais descritores de arquivo tenham sido redirecionados no momento da criação do processo para Portas de comunicação Conectáveis, a informação sobre essas portas é guardada nesta lista. Cada um de seus nós contém:
  - OMNIid da porta;
  - descritor de arquivo associado à porta;
  - nome simbólico da porta (caso ela tenha sido registrada no Servidor de Nomes);
- Descritor da conexão com o processo filho, usado para monitorar o comportamento de um processo criado pelo *daemon* até que ele efetue exec. **Não** é o mesmo que a

conexão de monitoramento presente na Lista de Processos Monitorados (LPMn), descrita abaixo.

O *daemon* possui também uma Lista de Processos Mortos (LPM). Esta lista contém informações sobre processos (possivelmente remotos) que já morreram e cujo pai ainda está vivo executando na máquina corrente. Cada nó da lista tem os seguintes campos:

- OMNlid do processo que morreu;
- Estado de saída desse processo;
- Índice do processo pai na TPC.

Outra estrutura do *daemon* é a Lista de Processos Monitorados (LPMn). Essa lista contém uma entrada para cada processo que não foi criado a partir de um `fork` do *daemon*. O objetivo da lista é permitir que o *daemon* detecte a morte desses processos. Cada entrada da lista possui:

- índice do processo na TPC;
- descritor de arquivo da conexão de monitoramento.

Por fim o *daemon* tem uma lista de processos locais bloqueados em uma chamada `om_wait`, conhecida como Lista de Processos em *Wait* (LPW). Cada nó da lista tem os campos abaixo:

- Índice do processo bloqueado em `om_wait` na TPC;
- Endereço TCP/IP (ou descritor da conexão) onde o processo está bloqueado esperando notificação da morte de seu filho.

6.2.1.3 Estruturas específicas da biblioteca

Através da biblioteca, o usuário pode ter acesso a uma ou mais Estruturas de Descrição de Processo (EDP), que armazenam todas as informações necessárias para especificar um processo a ser criado. A EDP contém todos os campos requeridos para guardar os atributos do processo descritos na seção 6.1.1.2, página 54. O usuário não precisa (e não deve) manipular os campos da EDP diretamente. Para isso são fornecidas as funções `om_initProc`, `om_copyProc`, `om_setProc` e `om_getProc`, descritas na seção 6.1. Além dos campos acessíveis ao usuário, a EDP tem também as seguintes informações, que não podem ser modificadas pelo usuário:

- Nome do usuário corrente;
- Identificação do usuário corrente (uid);
- Código indicando se EDP foi inicializada ou não, através de `om_initProc` ou `om_copyProc`.

A biblioteca mantém também, fora do alcance do usuário, um contador do número de filhos que foram criados mas para os quais não foi executada uma chamada a `om_wait`, ou seja, o Número de Filhos Vivos (NFV) do processo.

6.2.2 Criação de um Processo

A criação de um processo envolve os seguintes passos:

- O usuário inicializa uma EDP (Estrutura de Descrição de Processo) usando `om_initProc`. A EDP passa a conter, essencialmente, a descrição do processo corrente;
- Usando `om_setProc`, o usuário altera os valores dos atributos presentes na EDP conforme sua preferência, a fim de especificar corretamente o processo a ser criado;
- O usuário chama `om_forkExec`, passando a EDP como parâmetro;
- A biblioteca entra em contato com o *daemon* da máquina onde o novo processo irá residir e entrega a EDP a ele;
- O *daemon* confere se o usuário tem mesmo permissão de criar um processo naquela máquina, usando a mesma política de controle de acesso do comando `rsh`. O *daemon* autentica o usuário baseado apenas nos campos restritos da EDP. Se o usuário não tem permissão, a operação é terminada e `om_forkExec` retorna erro;
- O *daemon* executa um `fork`, criando um novo processo;
- Um canal de comunicação é aberto entre o *daemon* filho e o *daemon* pai. Esse canal será utilizado para comunicar ao pai a ocorrência de eventuais erros no filho ou, conforme veremos logo adiante, para enviar-lhe as OMNIids das portas criadas pelo filho. A *flag* de *close-on-exec* do canal é ligada (veja `fcntl(2V)` em [SRM90]),de forma que, caso não haja nenhum erro até o momento do `exec`, o pai receberá apenas fim-de-arquivo, pois a conexão será fechada automaticamente;
- O *daemon* pai põe-se a esperar a chegada de alguma informação pelo canal de comunicação com seu filho, ao passo que o filho prossegue executando;
- O *daemon* filho ajusta todos os atributos requeridos pelo usuário na EDP, mas ainda não é feito `exec`;
- Nesse momento existem duas possibilidades:
  - Caso algum descritor de arquivo tenha sido redirecionado para uma porta conectável de comunicação, não será possível efetuar o `exec` nesse momento. Ele só poderá ser feito depois de todas as portas terem sido conectadas através de `om_connect`. O *daemon* filho apenas testa se o arquivo a ser executado existe, se é acessível e se tem permissão de execução (o que não garante que o `exec` irá ter sucesso mais tarde, mas é o melhor que se pode fazer). Depois disso, cria as portas de comunicação e retorna suas OMNIids pela conexão com o *daemon* pai, mantendo ainda a conexão aberta por enquanto. Caso alguma das portas possua nome simbólico, ele a registra no Servidor de Nomes. Em seguida, fica bloqueado na função `om_waitConn` (veja seção 5.1.2), aguardando que algum outro processo conecte todas as suas portas;
  - Caso nenhum descritor de arquivo tenha sido redirecionado para uma porta conectável, o *daemon* filho efetua `exec` a fim de "reencarnar" no novo processo;
- O *daemon* pai recebe pela conexão com o filho a informação por ele enviada. Se algum erro tiver ocorrido, essa informação é passada ao processo chamador e `om_forkExec` retorna o erro. Caso tudo tenha corrido bem, uma nova entrada na TPC (Tabela de Processos Correntes) é criada para esse processo;

- A OMNIid do novo processo, bem como as OMNIids das portas de comunicação para as quais seus descritores de arquivo foram redirecionados (caso algum tenha sido), são retornadas à biblioteca no processo chamador;
- A biblioteca incrementa o contador com o Número de Filhos Vivos (NFV);
- A primitiva `om_forkExec` retorna ao usuário a OMNIid do processo recém criado e de suas portas (caso haja alguma).

Caso algum dos descritores de arquivo do processo tenha sido redirecionado para uma Porta Conectável, o novo processo irá aguardar que suas portas sejam conectadas por algum outro processo. O uso mais comum do sistema é que o próprio processo pai conecte as portas de seu filho logo após tê-lo criado.

Assim que o filho tiver todas as suas portas conectadas, ele irá efetuar `exec` e transformar-se em outro processo, que herdará todas as conexões estabelecidas. Nesse momento é possível que a chamada a `exec` falhe, e é necessário prover uma maneira do usuário tomar conhecimento desse erro. Note que o processo pai já retornou da chamada a `om_forkExec`, portanto não é possível avisá-lo através de um código de retorno dessa função. Nesse caso, o usuário é informado do erro através de `om_wait`, que retorna a OMNIid do processo juntamente com um código de erro avisando que não se conseguiu efetuar o `exec` com sucesso. Para implementar isso, o seguinte procedimento é adotado:

Após retornar sem sucesso do `exec`, o processo filho envia uma mensagem através da conexão com o pai informando-o do erro e encerra a conexão. Em seguida, efetua `exit` com um valor de saída igual a 254 (esse valor é arbitrário, qualquer outro poderia ser usado). Ao perceber que o filho terminou com código de saída 254, o *daemon* "suspeita" que o `exec` falhou. Para confirmar se é esse o caso, ele lê da conexão com seu filho. Se obtiver o código de falha no `exec`, sabe que o erro de fato ocorreu. Se receber apenas fim-de-arquivo, é porque o `exec` havia dado certo e foi o próprio processo do usuário que terminou com código 254, não havendo portanto nenhum erro.

6.2.3 Inicialização do Processo

Um processo comum (veja seção 6.1.5), jamais irá se inicializar junto ao OMNI e não utilizará de nenhuma de suas primitivas. Entretanto, se tiver seus descritores de arquivo redirecionados a portas de comunicação, estará automaticamente integrado a alguma computação distribuída OMNI.

Um processo OMNI, contudo, deve inicializar-se usando `om_init` antes de usar qualquer primitiva OMNI. Além dos efeitos explicados em outros capítulos, a chamada a essa função fará com que o processo entre em contato com o *daemon* Gerenciador local à sua máquina. Nesse momento, várias providências são tomadas:

Se o processo não foi criado a partir de um `fork` do *daemon*, como é o caso, por exemplo, de processos disparados na linha de comando da *shell*, é criada para ele uma entrada na TPC (Tabela de Processos Correntes). Um problema com esses processos é

que não há nenhuma maneira imediata do *daemon* perceber sua morte, uma vez que não foi ele que o criou. Para contornar essa situação, o seguinte procedimento é usado:

Uma conexão de monitoramento é criada entre o processo em inicialização e o *daemon*. Uma entrada para esse processo é colocada na LPMn (Lista de Processos Monitorados). O *daemon* ajusta seu lado da conexão para lhe enviar um sinal (SIGIO ou SIGPOLL, conforme o sistema seja BSD ou *System V*) quando algo chegar por ela. A biblioteca, por sua vez, nunca envia nada pela conexão. Quando o processo morre, o UNIX automaticamente fecha todos os seus descritores de arquivo, em particular o da conexão de monitoramento. Isso faz com que um fim-de-arquivo seja enviado ao *daemon*, que é interrompido pelo sinal e assim detecta a morte do processo.

Ainda durante a inicialização, caso algum dos descritores de arquivo do processo tenham sido redirecionados para portas de comunicação por seu pai, uma lista contendo a OMNIid, descritor associado e nome simbólico (caso exista) de cada uma das portas é extraída da TPC e retornada pelo *daemon*. Essa lista é usada pela biblioteca para atualizar as estruturas do módulo de portas, entre elas a TP (Tabela de Portas), e depois retornada ao usuário que ativou `om_init`.

Por fim, o contador NFV (Número de Filhos Vivos) é inicializado com zero.

6.2.4 Envio de sinais

Para enviar um sinal a um único processo, o usuário deve, de posse de sua OMNIid, utilizar a função `om_kill`. A biblioteca então entra em contato com o *daemon* onde reside o processo e pede que ele envie o sinal. Caso não haja problemas, o sinal é enviado e um código indicando sucesso é retornado ao usuário.

Caso se deseje enviar um sinal a um grupo de processos, um *broadcast* é feito na rede, enviando uma mensagem que requisita a todas as máquinas que enviem o sinal aos processos daquele grupo. Cada máquina checa em sua TPC (Tabela de Processos Correntes) se existe algum processo pertencente ao grupo. Em caso afirmativo (e caso haja permissão para isso), o sinal é enviado. Cada uma das máquinas responde ao *broadcast* dizendo quantos de seus processos receberam o sinal. Se nenhuma delas tiver conseguido enviá-lo para pelo menos um processo, a função `om_killpg` retorna erro. Caso contrário, é bem sucedida.

Conforme dito anteriormente (veja seção 6.1.2.1, página 57), um processo líder de sessão pode requisitar que, no momento de sua morte, um sinal de *hangup* seja enviado a todos os membros da sessão. A função que habilita o envio do sinal é `om_enableSighup`. Quando a morte de tal líder é detectada, um *broadcast* é feito na rede requisitando que todas as máquinas que possuam processos pertencentes àquela sessão lhes enviem SIGHUP. Cada *daemon* então executa essa ordem, mas não há necessidade de responder ao *broadcast*, pois não haveria para quem informar uma condição de erro.

6.2.5 Morte de um Processo

Quando um processo termina, esse fato é imediatamente percebido pelo *daemon* Gerenciador de sua máquina. O *daemon* detecta a morte de um processo por ele criado através da chegada de um sinal SIGCHLD. Ao recebê-lo, desvia para o tratador de sinal e descobre qual processo morreu usando a chamada ao sistema `wait`. Caso o processo não tenha sido criado pelo próprio *daemon*, ele estará sendo monitorado (de acordo com o explicado na seção 6.2.3) e será recebido um sinal SIGIO ou SIGPOLL (conforme o sistema seja BSD ou *System V)*. Uma vez nesse tratador, o *daemon* descobre qual processo morreu usando a informação da LPMn (Lista de Processos Monitorados).

Caso o processo que acaba de morrer seja um líder de sessão que requisitou o envio de um sinal de *hangup* a todos os membros na ocasião de sua morte, o sinal é enviado, conforme explicado na seção 6.2.4.

Quando é detectada a morte de um processo criado por uma primitiva OMNI, o *daemon* local primeiramente verifica se o processo terminou com código de saída 254. Conforme explicado na seção 6.2.2, o `exec` de um processo que teve seus descritores redirecionados para Portas Conectáveis pode falhar, caso em que o processo termina com código de saída 254. Nesse momento, o *daemon* deve confirmar se o `exec` realmente falhou ou não, através de uma consulta à conexão com seu filho. Em ambos os casos, ele envia para o *daemon* da máquina onde reside processo pai uma mensagem contendo:

- a OMNIid do processo que morreu;
- o estado de saída do processo (conforme retornado por `wait`);
- a identificação única (uqid) do processo pai;
- um código indicando se o `exec` do processo falhou ou não.

O *daemon* do pai, de posse dessas informações, verifica em sua TPC (Tabela de Processos Correntes) se o pai continua vivo. Em caso afirmativo, essas informações são inseridas na LPM (Lista de Processos Mortos) e poderão ser obtidas pelo pai quando ele executar `om_wait`. Caso contrário, essas informações são simplesmente descartadas.

O *daemon* do pai vasculha também sua LPW (Lista de Processos em *Wait*) para ver se o pai do processo que morreu não está bloqueado em `om_wait` esperando um de seus filhos morrer. Em caso afirmativo, a informação é enviada ao pai, liberando-o e fazendo com que ele retorne da chamada a `om_wait`. Note que, caso o exec do filho tenha falhado, `om_wait` retornará esse erro, juntamente com a OMNIid do processo.

O *daemon* da máquina do processo que terminou também checa sua LPM (Lista de Processos Mortos), pois lá pode existir entradas referentes a filhos já mortos do processo que acabou de morrer. Essas entradas são retiradas da LPM e descartadas, pois o único processo que podia efetuar `om_wait` para recuperá-las já morreu.

O *daemon* da máquina do processo que morreu faz também uma chamada ao Servidor de Nomes e remove todos os registros de entidades referentes àquele processo.

6.2.6 Espera Pela Morte de um Processo

Quando a primitiva `om_wait` é acionada, a biblioteca primeiramente consulta o valor de NFV (contador do Número de Filhos Vivos) para ver se o processo ainda tem algum filho pelo qual possa esperar. Em caso afirmativo, a biblioteca entra em contato com o *daemon* local, que vasculha a LPM (Lista de Processos Mortos) à procura de um filho do processo chamador que já tenha morrido. Caso encontre algum, retorna a informação ao pai imediatamente. Se nenhum filho morto é encontrado, uma entrada para o pai é inserida na LPW (Lista de Processos em *Wait*) e o pai é deixado esperando na conexão até que um de seus filhos termine.

Quando um de seus filhos finalmente morre, o pai recebe a informação sobre ele e retorna da chamada a `om_wait`, conforme explicado na seção 6.2.5. Antes de retornar, contudo, a biblioteca decrementa o valor do NFV (Número de Filhos Vivos).

Note que o uso de um único contador para o Número de Filhos Vivos faz com que `om_wait` não seja tolerante a falhas das máquinas onde residem os processos filhos. Imagine o seguinte cenário:

- O usuário efetua `om_wait`;
- Como NFV é maior que zero, a biblioteca entra em contato com o *daemon* local e o processo fica bloqueado esperando que algum de seus filhos morra;
- A máquina onde seu filho está executando falha e é reinicializada;
- Como nunca chegará nenhum aviso da máquina que falhou sobre a morte do filho que está sendo esperado na máquina local, o processo fica bloqueado em `om_wait` indefinidamente.

Para solucionar esse problema, a biblioteca teria que manter um registro mais preciso sobre quais e quantos de seus processos estão executando em quais máquinas e teria que ser notificada pelo *daemon* no caso de falhas. Dessa forma seria possível saber quais processos foram destruidos pelas falhas e permitir que `om_wait` retornasse erro para eles.

6.2.7 Prevenção de *Deadlocks*

Conforme mostrado ao longo do capítulo, existem situações onde dois *daemons* do Gerenciador podem simultaneamente tomar a iniciativa de se comunicarem. Assim sendo, o Gerenciador de Processos sofre dos mesmos problemas de *deadlock* discutidos na seção 4.4.8 para o Servidor de Nomes. As mesmas soluções lá descritas aplicam-se também ao seu caso.

Capítulo 7

Implementação do Protótipo do Sistema

A descrição do Sistema OMNI apresentada nos capítulos anteriores é o resultado de refinamentos sucessivos de idéias. Fundamental nesse processo de refinamento foi a implementação de um protótipo do sistema, que nos permitiu por à prova, na prática, várias das idéias e sentir suas deficiências, levando a uma melhor compreensão dos problemas envolvidos e de quais seriam suas soluções.

O protótipo é uma versão simplificada do sistema, não implementando toda a funcionalidade constante da atual especificação. Abaixo expomos os recursos oferecidos por cada módulo:

- **Servidor de Nomes**: toda a funcionalidade do Servidor foi implementada, entretanto isso foi feito de forma centralizada utilizando o sistema de arquivos, ficando a distribuição por conta do NFS (*Network File System*) [NPG90]. A nível do usuário, não é possível perceber a centralização, pois não foi feita nenhuma alteração na interface ou no modo de usá-lo;
- **Módulo de Portas**: apenas as portas conectáveis foram implementadas. Os conectores especiais foram implementados fora do núcleo do OMNI e possuem algumas limitações, como número fixo (porém configurável) de conexões com outras portas. Não chegaram a ser implementados conectores locais;
- **Gerenciador de Processos**: é capaz somente de criar processos remotos e enviar sinais a eles. O Gerenciador foi integrado ao Servidor de Nomes, mas não ao Módulo de Portas. A integração com esse módulo, que permitiria o redirecionamento de descritores de arquivo para portas de comunicação foi, contudo, simulada fora do núcleo OMNI, a fim de avaliar sua adequação.

O protótipo, que possui cerca de 10.000 linhas de código em C, foi implementado inicialmente em SunOS 4.1 (BSD) e posteriormente portado para Solaris 2.2 e SCO Open DeskTop (ambos *System V*).

7.1 Aplicações Experimentais

Alguns programas distribuídos foram implementados sobre o protótipo do OMNI, a fim de avaliar sua adequação. Podemos destacar dois deles:

MergeSort
Esta é uma simplificação do programa distribuído `MergeSort`, dado como exemplo na seção 2.4. Neste programa, um conector *mailbox* lê o conteúdo de um arquivo e o distribui a duas instâncias do programa `sort` do UNIX, que estão rodando em máquinas distintas. Essas ordenações parciais são unidas pelo programa `merge`, que as envia para um arquivo.

TeamSim
Implementado por Carlos Alberto Furuti, este é um simulador gráfico e sonoro de veículos que permite que o usuário pilote um carro, avião, helicóptero ou tanque de guerra. Cada participante, sentado em seu próprio computador, vê o cenário e os demais veículos dos outros usuários a partir de seu próprio ponto de vista, podendo persegui-los, atacá-los ou interagir com eles como quiser. Todos os processos envolvidos na simulação são criados e comunicam-se através do OMNI. O TeamSim foi implementado em Solaris 2.2 e portado para SCO Open DeskTop e SunOS 4.1, sendo possível, portanto, ter um participante pilotando seu veículo em um PC 486 e outro numa SPARCstation.

Apesar do TeamSim ter sido implementado sobre X-Window, que permite a criação de janelas em qualquer máquina da rede, apenas janelas locais são usadas. A computação da cena vista por cada usuário é efetuada em sua máquina local e a distribuição é proporcionada pelo OMNI.

As aplicações contribuíram bastante para expor os problemas do sistema e apontar as direções que devíamos tomar para melhorá-lo. Esse fato é melhor discutido na seção 7.3

7.2 OMNI nos cursos de Graduação

O autor deste trabalho atuou do segundo semestre de 1991 ao primeiro de 1994 como auxiliar didático das disciplinas de graduação MC 505 (Laboratório de Sistemas Operacionais) e MC 705 (Laboratório de Programação de Sistemas), ministradas por seu orientador. Estas são disciplinas práticas, oferecidas alternadamente a cada semestre, sendo uma pré-requisito da outra.

A cada semestre, era apresentada uma especificação funcional simplificada de um dos módulos do OMNI, que definia a interface e o comportamento de suas primitivas. Não era dito aos alunos *como* essas funções poderiam ser implementadas, ficando a cargo deles descobrir isso e gerar uma especificação interna do módulo. Uma vez que elaborar essa especificação é não-trivial, eles eram amparados nessa tarefa pelo professor e seu auxiliar. Uma vez fechada a especificação, eles partiam para sua implementação.

Esta foi uma experiência muito positiva para os alunos, que foram expostos a problemas complexos de computação distribuída, alguns com soluções ainda em aberto, sendo forçados a analisá-los com cuidado e especificar muito bem antes de começar a programar. A implementação exigiu também que adquirissem bom conhecimento do sistema UNIX e suas ferramentas, expondo os alunos à melhor tecnologia disponível nessa área. Alguns alunos foram inclusive capazes de criar soluções inéditas para alguns problemas ou detectar deficiências que não haviam sido percebidas, contribuindo assim para elaboração da especificação final do OMNI.

7.3 Resultados

Apesar das simplificações impostas ao protótipo, acreditamos que sua implementação em SunOS, seguida do porte para Solaris e SCO ODT, bem como a elaboração de algumas aplicações experimentais, em conjunto com a experiência adquirida ao expor os alunos de graduação ao sistema, permitiu que fossem elucidadas muitas das dificuldades e necessidades inerentes a um sistema do porte e natureza do OMNI. Acreditamos também que o *know-how* adquirido nessa experiência deverá ser valioso para a continuação do desenvolvimento de ferramentas para sistemas distribuídos no Projeto A_HAND.

Dentre as conclusões obtidas nesse processo, podemos destacar:

- O OMNI é um sistema grande, maior e mais complexo do que havíamos antecipado inicialmente. Para especificá-lo e implementá-lo num tempo razoável, seria necessário mais de uma pessoa;
- A implementação do sistema se beneficiaria imensamente de recursos de *multithreads*, caso eles estivessem disponíveis no ambiente. Além de aumentar a eficiência dos *daemons*, eles ajudariam a resolver os problemas de *deadlock* decorrentes do fato de dois *daemons* poderem simultaneamente tomar a iniciativa de se comunicarem um com o outro (ou de vários *daemons* formarem um ciclo). Maiores detalhes sobre os problemas de *deadlock* são dados na seção 4.4.8;
- Na implementação do protótipo, utilizamos sinais (SIGIO ou SIGPOLL) para interromper o processo quando alguém queria conectar suas portas. Dessa forma, quando alguém enviasse uma ordem de conexão, o processo seria desviado para o tratador, efetuaria a conexão da porta e retomaria o fluxo de execução anterior. Isso faria com que o processo que quer conectar a porta e o dono da porta não tivessem que bloquear esperado um *rendevouz* um com o outro. Percebemos que isso foi um erro, pois os sinais, além de causarem dificuldades no porte para *System V*, cujas primitivas são um pouco diferentes, eram eventos intrusivos no processo, pelos seguintes motivos:
  - Aplicações com restrições fortes de tempo (como aplicações de tempo real ou altamente interativas) podem ter suas execuções desviadas para o tratador durante muito tempo, o que pode causar pausas inaceitáveis em seu processamento;
  - Muitas chamadas ao sistema operacional (principalmente no *System V*) retornam erro quando interrompidas por um sinal (`errno == EINTR`), forçando o usu-

ário a testar a cada chamada passível deste erro se ele ocorreu ou não e, caso tenha ocorrido, efetuar novamente a chamada;

- O fato do OMNI capturar os sinais de SIGIO ou SIGPOLL impede que o usuário se utilize deles para suas aplicações.

- Durante a elaboração das aplicações, sentiu-se necessidade de algumas facilidades que não havíamos previsto. Entre elas:
  - Os conectores especiais locais, que antes não existiam e cuja necessidade foi sentida quando cogitamos a possibilidade de modificar os fontes do programa `make` do Projeto GNU para permitir que regras independentes fossem executadas paralelamente em máquinas diferentes. Nesse caso, os resultados das compilações em cada máquina deveriam ser mostrados no terminal local. Sem conectores locais, isso fica mais complicado;
  - Portas Conectáveis precisariam permitir, opcionalmente, mais de uma conexão por porta. Essa necessidade foi sentida quando implementamos os conectores especiais fora do núcleo do OMNI. As portas dos conectores são especiais justamente por permitirem mais de uma conexão. Como essa facilidade não estava presente fora do núcleo OMNI, um processo a nível de usuário não tem como implementar um conector especial da maneira como ele é definido;
  - Seria interessante integrar o OMNI a uma facilidade como XDR (*eXternal Data Representation*) [NPG90] ou algo similar para permitir enviar estruturas tipadas de forma independente de máquina. Esse problema surgiu quando fomos interligar duas instâncias do TeamSim executando em máquinas de arquiteturas diferentes. A solução usada foi converter os dados em *strings* ASCII antes de enviá-los (vale a pena ressaltar que programas escritos em Cm distribuído não sofreriam esse problema).

Capítulo 8 Conclusões

O sistema OMNI oferece uma base sobre a qual aplicações distribuídas podem ser construidas com maior rapidez e confiabilidade. Conforme mostrado na seção 7.3, a menos de alguns ajustes ainda necessários, o sistema mostrou-se adequado aos seus propósitos em testes preliminares.

O Servidor de Nomes provê um mecanismo de armazenamento de informações projetado para atender às necessidades das demais camadas do OMNI, porém passível de ser utilizado independentemente delas. O Servidor inclui aspectos de tolerância a falhas, pois faz com que as informações sobre entidades locais sejam armazenadas localmente. Assim, quando uma máquina falha, a tanto as entidades quanto seus registros no Servidor são perdidos, evitando que seja necessário que alguma outra máquina perceba a falha da primeira e remova a informação das entidades a ela associadas. Isso é feito sem prejudicar o desempenho, pois as informações são mantidas em *caches* nas máquinas que fazem uso mais intenso delas.

O Gerenciador de Processos provê serviços muito similares aos encontrados no UNIX, estendidos a um ambiente distribuído. Esse fato beneficia os programadores já acostumados ao ambiente UNIX, ao mesmo tempo que oferece aos usuários em geral um alto grau de controle sobre seus processos.

O mecanismo de comunicação oferecido pelo Módulo de Portas, além de eficiente, permite que processos conectem portas que não lhe pertencem, bem como suas próprias. Isso possibilita a criação de computações distribuídas complexas através da interconexão das portas dos processos componentes, que não precisam ter nenhum conhecimento da estrutura global da computação. Através da utilização de conectores especiais, tais computações podem fazer com que os dados sejam disseminados a grupos de processos.

Uma vez que o sistema é capaz de criar e comunicar processos comuns (isto é, que não têm conhecimento do OMNI), uma vasta gama de programas já existentes no UNIX pode ser incluída em computações distribuídas.

O OMNI contribuiu para ampliar os conhecimentos na área de sistemas distribuídos, não apenas do autor deste trabalho, mas da equipe do Projeto A_hand como um todo. Além, é claro, dos problemas teóricos por ele levantados (muitos deles largamente discutidos na literatura), vários problemas de ordem prática tiveram também que ser enfrentados, como, por exemplo, aprofundamento dos conhecimentos sobre o sistema operacional UNIX e Sistemas Abertos, domínio da tecnologia de comunicação entre processos e outras ferramentas disponíveis no UNIX, problemas de portabilidade entre várias plataformas, problemas de engenharia de *software* envolvidos na coordenação de várias pessoas trabalhando simultaneamente em componentes relacionados (OMNI-*LegoShell*-Cm Distribuído), etc.

8.1 Direções Futuras

O OMNI ainda não está devidamente integrado ao Cm Distribuído. A curto prazo, esta deverá ser a próxima meta. Isso nos trará um melhor entendimento das dificuldades e necessidades inerentes ao oferecimento de suporte a um sistema distribuído a nível de *objetos*.

A médio prazo, o Projeto A_hand e outros membros do DCC/UNICAMP deverão unir-se ao LCMI/UFSC, DEE/PUC-Rio e DEE/UFC no desenvolvimento do ASAP [ASP94]: Ambiente para Suporte a APlicações distribuídas baseado em objetos, cuja proposta foi recentemente aprovada pelo ProTem-CC. Este é um ambiente muito mais ambicioso do que o OMNI e que, entre outras coisas, deverá prover:

- Suporte a objetos distribuídos, incluindo:
  - ativações remotas de métodos;
  - transparência quanto a localização dos objetos;
  - objetos *multithreaded*;
- Facilidades para implantação de tolerância a falhas;
- Suporte a aplicações de tempo real.

Estas facilidades deverão ser implementadas tendo em vista requisitos de portabilidade e conformidade a padrões *de facto*, como UNIX e TCP/IP, e emergentes, como DCE e CORBA.

O OMNI e o ambiente RIO [WL93], desenvolvido pelos pesquisadores da PUC-Rio, deverão ser revistos em função dessa nova abordagem, e a experiência com eles obtida, bem como a derivada das linguagens do ambiente A_hand (Cm, *LegoShell* e CO^2), será incorporada ao ASAP. Com isso esperamos criar uma base sólida de suporte ao desenvolvimento de aplicações distribuídas robustas, oferecendo as ferramentas necessárias para que o programador se beneficie do paradigma de orientação a objetos nesse ambiente.

Apêndice A

Referências Bibliográficas

1. **Hn78** — **Distributed Processes: A Concurrent Programming Concept**
Per Brinch Hansen
Communications of the ACM, vol 21, nº 11, novembro de 1978.

2. **Hr78** — **Communicating Sequential Processes**
C. A. R. Hoare
Communications of the ACM, vol 21, nº 8, agosto de 1978.

3. **KR78** — **The C Programming language**
Brian W. Kernighan e Dennis M. Ritchie
Prentice–Hall Inc., 1978.

4. **Ch84** — **The V Kernel: A Software Base for Distributed Systems**
David R. Cheriton
IEEE Software. Abril de 1984.

5. **TLC85** — **Preemptable Remote Execution Facilities for the V-System**
Marvin M. Theimer, Keith A. Lantz e David R. Cheriton
ACM-0-89791-174-1-12/85-0002, 1985.

6. **ABG86** — **Mach: A New Kernel Foundation for UNIX Development**
Mike Accetta, Robert Baron, David Golub, et. al.
Relatório técnico, agosto de 1986.

7. **JR86** — **Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems**
Michael B. Jones e Richard F. Rashid
OOPSLA'86 Proceedings, setembro de 1986. ACM 0-89791-204-7/86/0900-0067.

[8] Lc86 **Introdução à Criptografia Computacional**
Cláudio Leonardo Lucchesi
Editora da Unicamp, 1986.

[9] DL87a **A-HAND: Ambiente de desenvolvimento de *software* baseado em Hierarquias de Abstração em Níveis Diferenciados**
Drummond, R. e Liesenberg, H.
IV Encontro de Trabalhos do Projeto ETHOS, Petrópolis, RJ, abril 1987.
Revisto e reimpresso como relatório técnico, Projeto A_HAND, outubro 1987.

[10] DL87b **Requisitos para um ambiente de desenvolvimento de Programas**
Rogério Drummond e Hans Liesenberg
I Encontro IBM de Ciência e Tecnologia em Informática, Rio de Janeiro, RJ, novembro 1987.

[11] DO87 **Process Migration in the Sprite Operating System**
Fred Douglis e John Ousterhout
Proccedings of the 7th International Conference on Distributed Computing Systems, Berlin, West Germany, 21-25 de setembro de 1987.

[12] BLD88 **Programação em Cm**
F. Q. B. da Silva, Hans K. E. Liesenberg e Rogério Drummond
Anais do XV Semish, pp 101–102, Rio de Janeiro, RJ, julho 1988.

[13] Ch88 **The V Distributed System**
David R. Cheriton
Communications of the ACM. Vol 31, nº 3, março de 1988.

[14] Co88 **Internetworking With TCP/IP**
Douglas E. Comer
Prentice Hall, 1988. ISBN 0-13-470154-2 025.

[15] KR88 **The C Programming language**
Brian W. Kernighan e Dennis M. Ritchie
2ª edição, Prentice–Hall Inc., 1988.

[16] OCD88 **The Sprite Network Operating System**
John K. Ousterhout, ASndrew R. Cherenson, Frederick Douglis, et. al.
Computer, fevereiro de 1988.

[17] AGH89 **Revolution 89 or "Distributing UNIX Brings it Back to its Original Virtues"**
François Armand, Michel Gien, Frédéric Herrmann e Marc Rozier
Proceedings of "Workshop on Experiences with Building Distributed (and Multiprocessor) Systems", 5-6 de outubro, 1989, Ft. Lauderdale, FL, USA, pp. 153-174.

[18] Dr89 ***LegoShell*– Linguagem de Computações**
Rogério Drummond
III Simpósio Brasileiro de Engenharia de Software, páginas 1-13, Recife, PE, outubro de 1989.

19. **BC90** **The Isis Project: Real experience with a fault tolerant programming system**
Kenneth Birmam e Robert Cooper
Relatório técnico TR 90-1138, Department of Computer Science, Cornell University, 1990.

20. **DCE90** **Distributed Computing Environment Overview**
Open Software Foundation, 1990.

21. **LWP90** **Programming Utilities & Libraries**
Sun Microsystems, Inc.
Manual técnico, 1990. Capítulo 2 "*Lightweight Processes*".

22. **NPG90** **Network Programming Guide**
Sun Microsystems, Inc.
Manual técnico, 1990.

23. **RAA90** **Overview of the CHORUS® Distributed Operating Systems**
M. Rozier, V. Abrossimov, F. Armand, et.al.
Relatório técnico CS/TR-90-25, Chorus Systèmes, abril de 1990.

24. **SNA90** **System and Network Administration**
Sun Mycrosystems, Inc.
Manual técnico, 1990.

25. **SRM90** **SunOS Reference Manual**
Sun Mycrosystems, Inc.
Manual técnico, 1990.

26. **FD91** **A_HAND – Ambientes e Linguagens**
Maurício Fernandez e Rogério Drummond
Relatório técnico, Projeto A_HAND - DCC - IMECC - UNICAMP, capítulo "Sobre uma linguagem de prototipagem para ambiente UNIX", 1991.

27. **Fr91** **Um compilador para uma linguagem de programação orientada a objetos**
Carlos Alberto Furuti
Tese de mestrado, DCC-IMECC-UNICAMP, julho 1991.

28. **MDK91a** **Constructive Communication in MP**
Jeff Magee, Naranker Dulay e Jeff Kramer
Realtório técnico, department of Computing, Imperial College of Science, Technology and Medicine.

29. **MDK91b** **Structuring Parallel and Distributed Programs**
Jeff Magee, Naranker Dulay e Jeff Kramer
Realtório técnico, department of Computing, Imperial College of Science, Technology and Medicine.

30. **ORB91** **The Object Management Group Object Request Broker RFP Joint Response**
Hewlett-Packard Company e Sun Microsystems, Inc., 1991.
OMG Document Number: 91.1.4.8, Part No: 800-6274-01.

31. **CB92** **Using the ISIS Resource Manager for Distributed, Falut-Tolerant Computing**
Timothy Clark e Kenneth Birman
Relatório técnico, Department of Computer Science, Cornell University, junho de 1992.

32. **COR92a** **The Common Object Request Broker: Architecture and Specification**
Object Management Group e X/Open
OMG Document Number: 91.12.1. John Wiley & Sons, Inc., 1992, ISBN 0-471-58792-3.

33. **COR92b** **Object Management Architecture Guide**
Object Management Group
OMG TC Document 92.11.1. John Wiley & Sons, Inc., 1992, ISBN 0-471-58563-7.

34. **DC92** **OMNI – Sistema de suporte a aplicações distribuídas**
Rogério Drummond e Cassius Di Cianni
Anais do VI Simpósio Brasileiro de Engenharia de Software, pp 309–324, novembro 1992.

35. **MD92** **MP: A Programming Environment for Multicomputers**
Jeff Magee e Naranker Dulay
Realtório técnico, department of Computing, Imperial College of Science, Technology and Medicine.

36. **Sh92** **Guide to Writing DCE Aplications**
John Shirley
O'Reilly & Associates, Inc., 1992. ISBN 1-56592-004-X.

37. **Br93** **The Process Group Approach to Reliable Distributed Computing**
Kenneth P. Birman
Relatório técnico TR 91-1216, Department of Computer Science, Cornell University, março de 1993.

38. **DCE93** **OSF™ DCE User's Guide and Reference**
Open Software Foundation
Prentice Hall, 1993. ISBN 0-13-643842-3.

39. **DGT93** **Aspectos da Implementação de Objetos Distribuídos**
Rogério Drummond, Celso Gonçalves Júnior e Alexandre P. Teles
Anais do 11° Simpósio Brasileiro de Redes de Computadores, pp 139–152, maio 1993.

40. **MT93** **Guide to Multi-Thread Programming**
Sun Microsystems, Inc.
Manual técnico do SunOS 5.3 (Solaris 2.3), 1993.

41. **Tl93** **A Linguagem de Programação Cm**
Alexandre P. Teles
Tese de mestrado, DCC - IMECC - UNICAMP, outubro de 1993.

42. **TLI93** **Network Interfaces Programmer's Guide**
Sun Microsystems
Manual técnico Slaris 2.3, maio de 1993.

[43] **WL93** **Ambiente RIO: Metodologia e Suporte para Sistemas Configuráveis**
J. Werner e O. Loques
XX SEMISH - Seminário Integrado de *Software* e *Hardware*, Florianópolis, setembro de 1993.

[44] **ASP94** **Um Ambiente para Suporte de Aplicações Distribuídas, baseado em objetos**
Vário autores, do DCC/UNICAMP, LCMI/UFSC, DEE/PUC-Rio e DEE/UFC
Proposta submetida e aprovada pelo ProTem-CC, março de 1994.

[45] **Fd94** **Concorrência em Linguagens de Comandos**
Mauricio Fernández
Tese de mestrado, DCC - IMECC - UNICAMP, agosto de 1994.

[46] **Gn94** **Objetos Distribuídos**
Celso Gonçalvez Júnior
Tese de mestrado, DCC - IMECC - UNICAMP, agosto de 1994.